DOCTORAT DE L'I.N.P.T.

Spécialité : **INFORMATIQUE ET TELECOMMUNICATIONS**

BEHNIA Salimeh

Laboratoire : **LABORATOIRE D'ANALYSE ET D'ARCHITECTURE DES SYSTEMES (LAAS - CNRS)**

Titre de la thèse :

TEST DE MODELES FORMELS EN B : CADRE THEORIQUE ET CRITERES DE COUVERTURE

Soutenance le :

27 OCTOBRE 2000 à 10 heures 30
Salle de Conférences du LAAS - CNRS de Toulouse

Directeur de recherche :

Mme Pascale THEVENOD-FOSSE

JURY :			
	Mme	Marie-Claude GAUDEL	Rapporteur
	M.	Bruno LEGEARD	Rapporteur
	M.	El Miloudi EL KOURSI	Examinateur
	M.	Cliff JONES	Examinateur
	M.	Luis-Fernando MEJIA	Examinateur
	Mme	Pascale THEVENOD-FOSSE	Directeur de thèse
	Mme	Hélène WAESELYNCK	Examinateur

MOTS CLES :

- Méthode B
- Test du logiciel
- Validation de modèles formels
- Oracle de test
- Analyse Structurelle
- Critères de couverture

Résumé

Les travaux présentés dans ce mémoire définissent un cadre théorique pour le test de logiciels développés selon la méthode formelle B. Les tests visent à révéler les fautes dues à une mauvaise compréhension ou à une mauvaise modélisation d'un besoin fonctionnel, et complètent ainsi les preuves effectuées pendant le développement formel.

Un développement B peut être vu comme une série d'étapes durant lesquelles des modèles de plus en plus concrets de l'application sont construits, le code final pouvant être considéré comme une version compilée du modèle le plus concret. Le cadre théorique de test que nous avons défini est un cadre unifié, indépendant du fait que les résultats de test soient obtenus de l'animation des modèles ou de l'exécution du code. Ce cadre est explicitement lié à la notion du raffinement des modèles B : pour une entrée de test, l'acceptation des résultats fournis par un modèle implique l'acceptation des résultats fournis par les raffinements corrects de celui-ci.

Nous définissons ensuite une approche d'analyse structurelle des modèles B. En poursuivant le cadre unifié, notre objectif est de définir des stratégies de couverture qui soient applicables à la fois à un modèle abstrait et à un modèle concret. Ceci a nécessité d'unifier pour les modèles B deux catégories de critères :

- critères de couverture des spécifications orientées modèle basés sur la couverture des prédicats avant-après ;
- critères classiques de couverture structurelle des programmes basés sur la couverture du graphe de contrôle.

A partir de cette unification, nous avons défini un ensemble de critères, ordonnés selon la relation d'inclusion, qui complètent les critères existants.

Mots clés : Méthode B, Test du logiciel, Validation de modèles formels, Oracle de test, Analyse structurelle, Critères de couverture.

Abstract

The work presented in this dissertation concerns the definition of a theoretical framework for testing software developed within the B formal method. The test aims to reveal specification faults due to a misunderstanding or a misrepresentation of a functional requirement, and thus complement the proofs performed during the formal development process.

The B development process can be seen as a series of steps during which successively more concrete models of the system are constructed, the final code being considered as a compiled version of the most concrete model. The theoretical framework that we have defined is a unified framework, independent of the fact that the results are obtained by animation of models or by execution of the final code. The framework is explicitly related to the notion of refinement of B models: for a given test input, the acceptance of the results of a given model implies the acceptance of the results of its correct refinements.

We then define an approach to structural analysis of B models. Following the unified framework, our aim is to define coverage strategies applicable to abstract models as well as to concrete ones. This has required the unification of two categories of criteria for B models:

- coverage criteria defined for model oriented specifications based on the coverage of before-after predicates;
- classical structural coverage criteria of programs based on the coverage of control flow graphs.

From this unification, we have defined a set of criteria, ordered according to the inclusion relation, that complete the existing hierarchy of criteria.

Keywords: B Method, Software Testing, Validation of Formal Models, Test Oracle, Structural Analysis, Coverage Criteria.

Avant-Propos

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS), dans le cadre d'une collaboration avec l'Institut National de Recherche sur les Transports et leur Sécurité (INRETS).

Je remercie Messieurs Alain Costes et Jean-Claude Laprie, qui ont successivement assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueillie au sein de ce laboratoire.

Je remercie également Messieurs Jean-Claude Laprie et David Powell, Directeurs de Recherche CNRS, responsables successifs du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

J'exprime ma profonde reconnaissance à Pascale Thévenod-Fosse, Directeur de Recherche CNRS, et Hélène Waeselynck, Chargée de Recherche CNRS, pour avoir dirigé mes travaux tout au long de cette thèse. J'ai apprécié leurs compétences, leur rigueur intellectuelle et l'enthousiasme avec lequel elles mènent leurs activités de recherche. Je remercie tout particulièrement Hélène Waeselynck, pour son travail d'encadrement assidu. Les travaux présentés dans ce mémoire ont largement bénéficié de ses qualités scientifiques et de ses conseils avisés. Qu'elle soit également remerciée pour son aide précieuse dans la rédaction de ce mémoire, pour sa grande disponibilité et sa patience.

Je tiens également à remercier Monsieur Gérard Couvreur, Ingénieur de Recherche INRETS et Directeur de l'unité de recherche Evaluation des Systèmes de Transports Automatisés et de leur Sécurité (ESTAS) de l'INRETS, pour m'avoir permis de réaliser cette thèse en collaboration avec son équipe. Je remercie vivement Monsieur El Miloudi El Koursi, Chargé de recherche INRETS, pour la confiance qu'il m'a témoignée durant mes travaux.

Je remercie Madame Pascale Thévenod-Fosse, Directeur de Recherche CNRS, pour l'honneur qu'elle me fait en présidant mon jury de thèse, ainsi que :

- Madame Marie-Claude Gaudel, Professeur à l'Université de Paris-Sud,
- Monsieur Bruno Legeard, Professeur à l'Université de Franche-Comté,
- Monsieur El Miloudi El Koursi, Chargé de recherche INRETS,
- Monsieur Cliff Jones, Professeur à l'Université de Newcastle au Royaume-Uni,
- Monsieur Luis-Fernando Mejia, Ingénieur chez Alstom Transport SA,
- Madame Hélène Waeselynck, Chargée de recherche CNRS,

pour avoir accepté de participer à ce jury et plus particulièrement Madame Marie-Claude Gaudel et Monsieur Bruno Legeard pour avoir accepté la charge d'être rapporteurs.

J'adresse mes remerciements à Monsieur Luis-Fernando Mejia, ingénieur chez Alstom Transport SA, qui, en me fournissant une étude de cas, m'a permis de mettre en pratique les concepts définis par mes travaux.

Je remercie Georges Mariano, Chargé de Recherche INRETS, et Dorian Petit, doctorant, pour l'intérêt qu'ils ont porté à mes travaux. Nos discussions sur l'implémentation d'outils autour de la méthode B ont été très enrichissantes.

Je remercie vivement tous les membres du groupe TSF, permanents, doctorants et stagiaires, et en particulier Yves Crouzet. Sa disponibilité et son aide technique m'ont souvent été d'un grand secours. J'associe bien sûr à ces remerciements Joëlle Penavayre et Marie-José Fontagne pour leur efficacité et pour leur collaboration dans l'organisation des derniers préparatifs en vue de la soutenance.

Le service Documentation-Edition et les différents services administratifs et logistiques du LAAS-CNRS, par leur efficacité et leur disponibilité, m'ont permis de travailler dans de très bonnes conditions. Je les en remercie sincèrement.

Ces avant-propos seraient incomplets sans un mot particulier pour mes amis Mathieu Robart et Cláudia Betous-Almeida qui ont toujours été à mes côtés surtout dans les moments difficiles. Qu'ils soient assurés de ma profonde reconnaissance et de mon éternelle amitié. Je tiens également à remercier Olfa Kaddour, Slim Abdellatif et Vincent Nicomette qui ont toujours eu un mot de réconfort à me dire quand j'en avais besoin. Mes amis sont trop nombreux pour être tous cités ici. Je les remercie tous en gardant une pensée particulière pour mes camarades de promotion Marc-Olivier Killijian, Yvan Labiche, Marc Mersiol et Mourad Rabah.

Enfin, je remercie ma famille pour son amour et soutien constant, même de loin.

Sommaire

Introduction générale

La société moderne devient sans cesse plus dépendante de l'informatique. Les systèmes actuels de transport, de télécommunications, d'administration et de soins médicaux ne pourraient plus fonctionner sans support informatique. Ce support doit souvent répondre à des exigences de sûreté de fonctionnement élevées, car ses défaillances peuvent avoir des conséquences catastrophiques en termes de pertes économiques ou même de vies humaines.

Réalisant des fonctions de plus en plus complexes, la partie logicielle des systèmes informatiques est susceptible d'être source de défaillances. Son développement demande alors la mise au point de techniques d'ingénierie et de validation adaptées, permettant d'assurer le niveau de sûreté de fonctionnement requis.

L'utilisation des méthodes formelles est recommandée pour le développement de logiciels critiques. Leur sémantique bien définie permet une spécification précise du service à délivrer. Elles sont fondées sur des bases mathématiques permettant éventuellement d'effectuer des preuves pour démontrer certaines propriétés de la spécification, ou encore de vérifier de manière formelle les phases de conception et d'implémentation. L'utilisation dans les projets industriels des méthodes formelles et des mécanismes de preuve qu'elles rendent possibles, connaît un succès grandissant [Bowen et Stavridou 1993; Hinchey et Bowen 1995; OFTA 1997; King et al. 1999].

En France, un exemple de méthode formelle utilisée avec succès en milieu industriel est la méthode B [Abrial 1996], notamment dans le domaine ferroviaire [Behm et al. 1997]. La méthode B est une méthode de développement, qui ne se limite pas à la phase de spécification. La spécification est raffinée jusqu'à l'obtention de modèles suffisamment concrets pour être directement traduits dans un langage de programmation. Cette chaîne de raffinement est accompagnée d'obligations de preuve qui sont générées automatiquement par la méthode.

Comme les autres techniques de vérification, les techniques de preuve ont leurs limites et ne permettent pas de garantir l'élimination de toutes les fautes [Hall 1990;

Bowen et Hinchey 1995; Gaudel 1995a]. Notamment, nous pouvons mentionner le problème de la vérification des modèles formels par rapport à l'expression informelle des besoins. Par essence, on ne peut pas prouver que les besoins informels de l'utilisateur sont correctement modélisés. D'autres techniques de vérification, comme le test, doivent être utilisées.

Les travaux rapportés dans ce mémoire portent sur la validation des logiciels développés selon la méthode B. Pour les applications industrielles de la méthode B, afin d'assurer que les besoins de l'utilisateur sont correctement pris en compte, on procède comme suit :

- vérification de la conformité des spécifications B par rapport au cahier des charges, par revue des sources B ;
- tests fonctionnels du code généré, d'abord sur machine hôte, puis en intégration avec le matériel et des simulateurs de l'environnement opérationnel.

La contribution de nos travaux porte sur le premier point, notre but étant de renforcer ces analyses manuelles par le test des modèles formels. Le test vise ici à révéler des fautes qui sont dues soit à une mauvaise compréhension d'un besoin fonctionnel, soit à une mauvaise modélisation d'un besoin bien compris. Pour s'avérer complémentaire aux preuves prescrites, le test doit tenir compte des spécificités du développement formel en B, notamment des mécanismes de raffinement.

Ce mémoire est structuré en quatre chapitres.

Dans le chapitre 1, nous introduisons les principaux concepts de la méthode B. Nous expliquons la notion de *machine abstraite*, qui constitue le modèle de base d'un développement en B, ainsi que le langage *des substitutions généralisées*, utilisé pour la spécification de la dynamique du système. Nous présentons ensuite la définition du raffinement des machines abstraites, les liens architecturaux permettant le développement incrémental de ces machines, et les obligations de preuve générées par la méthode.

Dans le chapitre 2, nous mettons en évidence la place du test dans un développement formel, et justifions l'intérêt de ces travaux menés dans le cadre d'un développement formel en B. Nous commençons par une synthèse des techniques de test structurel et fonctionnel de programme. Nous discutons ensuite de la complémentarité du test et de la preuve. Cette discussion nous amène à envisager l'utilisation des techniques de vérification dynamique, en plus des techniques statiques, dès les phases amont du développement. L'animation des spécifications est un moyen permettant d'effectuer de telles vérifications dynamiques. Nous donnons alors un bref aperçu des techniques et des outils d'animation des spécifications orientées modèle.

Dans le chapitre 3, nous présentons le cadre théorique de test que nous avons défini pour les modèles B. La définition de ce cadre s'appuie sur les caractéristiques de la méthode B que nous avons présentées dans le chapitre 1. Ce cadre théorique est indépendant du niveau d'abstraction des modèles B testés, et peut ainsi s'appliquer à n'importe quelle étape du développement formel. Il permet également de clarifier les notions de commandabilité et d'observabilité pour les modèles B. Nous définissons la

notion de séquence de test pour un modèle B en nous basant sur le comportement observable spécifié par celui-ci. La définition de l'oracle de test est compatible avec les obligations de preuve de raffinement de modèles.

Conformément au cadre théorique proposé, nous définissons dans le chapitre 4 des critères de test applicables à la fois aux modèles abstraits et concrets. Ces critères sont définis en termes de couverture structurelle de modèles B. Ils permettent d'établir un lien entre les approches classiques de couverture structurelle de programmes (analyse de graphe de contrôle) et celles proposées pour la couverture des spécifications orientées modèle (analyse de prédicats avant-après).

Introduction à la méthode B

1.1 Introduction

La méthode B, conçue par J.R. Abrial, est une méthode formelle qui couvre toutes les étapes de développement du logiciel, de la spécification jusqu'à la génération de code exécutable, par une succession d'étapes de raffinement. Elle est destinée à la conception de logiciels séquentiels, de taille modeste, qui manipulent des valeurs intégrales (entières, booléennes). Un développement B implique la démonstration des *obligations de preuves*, générées systématiquement par la méthode.

Ce chapitre vise à donner une vue générale de la méthode B, et à introduire les notions nécessaires à la compréhension des travaux développés dans ce mémoire. Il se base sur l'ouvrage de référence de la méthode B, le B-Book [Abrial 1996], qui fournit une présentation complète de la méthode et ses fondements théoriques.

Dans le paragraphe 1.2, nous présentons la notion de *machine abstraite* qui constitue la brique de base d'un développement B. Le paragraphe 1.3 est consacré au langage des substitutions généralisées, le langage de description de la dynamique d'un système modélisé en B.

L'architecture de développement d'un système logiciel en B est constituée de plusieurs niveaux de raffinement et de décomposition en couches. La notion de raffinement fait l'objet du paragraphe 1.4. Les mécanismes de structuration et de décomposition sont abordés dans le paragraphe 1.5. Le paragraphe 1.6 décrit les obligations de preuves prescrites par la méthode. Une brève description des outils commerciaux de développement en B est présentée dans le paragraphe 1.7. Enfin,

nous terminons dans le paragraphe 1.8 par quelques exemples d'application de la méthode B dans l'industrie ferroviaire française.

1.2 Machine abstraite

La méthode B, tout comme les langages de spécifications Z [Spivey 1994] et VDM [Jones 1993], appartient à la famille des spécifications orientées modèle. Dans cette approche, un modèle du système à construire est spécifié. Ce modèle est constitué des états que le système peut prendre, des relations d'invariance qui sont maintenues lorsque le système passe d'un état à un autre, et des opérations permettant de fournir un service aux utilisateurs du système. Ainsi, l'invariant représente les aspects statiques du système spécifié et les opérations ses aspects dynamiques. L'état n'est pas accessible de l'extérieur, et on ne peut donc y accéder ou le modifier qu'à travers les opérations.

La brique de base d'une spécification B est la machine abstraite. Une machine abstraite peut être vue comme une boîte noire ayant une mémoire et un certain nombre de boutons. Les valeurs stockées dans cette mémoire représentent l'état de la machine et les boutons sont des opérations que l'utilisateur de la machine peut activer. Ainsi, la machine abstraite contient la spécification des aspects statiques et des aspects dynamiques du système. Dans la syntaxe de la méthode B, les machines abstraites sont décrites de manière incrémentale dans des composants de type MACHINE, REFINEMENT et IMPLEMENTATION, en utilisant une notation unique appelée *Abstract Machine Notation* (*AMN*).

La Figure 1.1 montre l'exemple d'une machine abstraite modélisant une pile. Comme on peut le voir sur cet exemple, le texte d'un composant B est structuré en *clauses* (MACHINE, SEES, CONSTRAINTS, …).

On peut distinguer quatre parties dans un composant B : l'en-tête, les clauses de composition, une partie déclarative et une partie exécutive. Dans l'en-tête, on trouve le nom de la machine abstraite, suivi éventuellement d'une liste de paramètres. Les clauses de composition permettent d'établir des liens de construction avec d'autres composants. Dans l'exemple de la Figure 1.1, la machine Stack_m *voit* (clause SEES) la machine Object_m, pour pouvoir consulter ses constituants. Nous expliquerons dans le paragraphe 1.5 les différentes clauses de composition qui sont définies en B. La partie déclarative contient la définition de l'état et le contexte de la machine abstraite. Enfin, la partie exécutive contient la définition des opérations et l'initialisation.

Une machine abstraite, comme celle de la Figure 1.1, peut être paramétrée. Les propriétés de ces paramètres, par exemple les propriétés de typage, sont décrites dans la clause CONSTRAINTS. Il est également possible de définir des constantes et des ensembles pour enrichir le contexte de la machine. Les constantes sont définies dans la clause CONSTANTS, les ensembles dans la clause SETS et leurs propriétés dans la clause PROPERTIES (l'exemple de la Figure 1.1 ne contient pas de constante ni

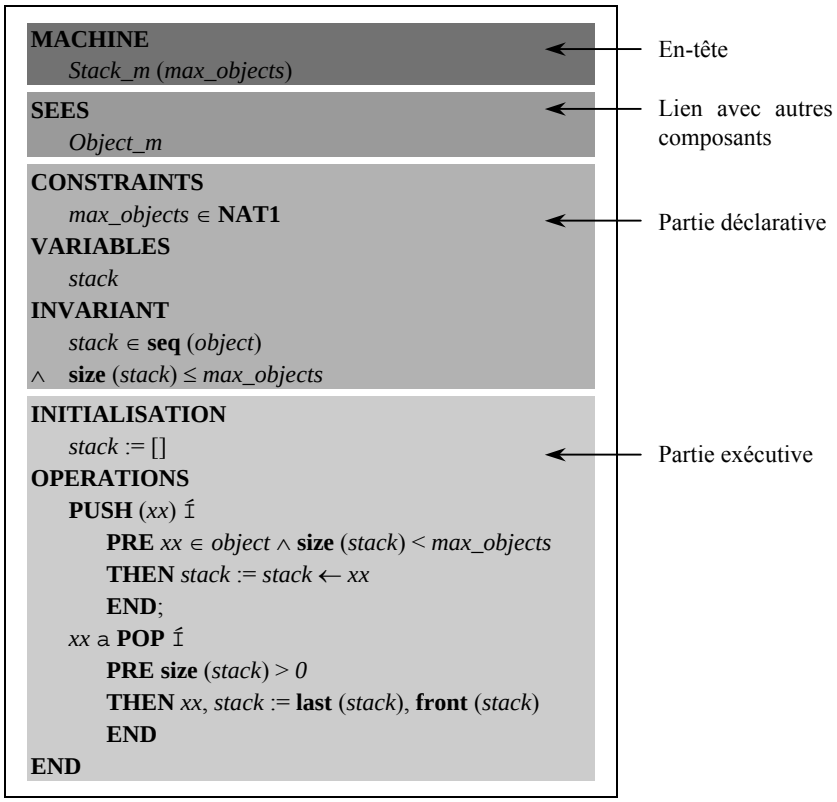


Figure 1.1. Exemple d'un composant de type MACHINE modélisant une pile

d'ensemble). L'état de la machine abstraite est modélisé au travers de variables d'état dans la clause VARIABLES et les propriétés d'invariance sont décrites dans la clause INVARIANT. Les variables d'état, les paramètres, les constantes et les ensembles, ainsi que leurs propriétés respectives, sont spécifiés en utilisant des formules de la logique du premier ordre et des concepts mathématiques de la théorie des ensembles.

L'initialisation et les opérations constituent la partie exécutive de la machine abstraite. La valeur initiale de l'état est spécifiée dans la clause INITIALISATION. La clause OPERATIONS contient la liste des opérations de la machine qui permettent d'accéder à l'état interne et de le modifier. Certaines opérations peuvent également retourner des valeurs de sortie, comme l'opération POP de la Figure 1.1.

Pour la spécification des opérations et de l'initialisation, la méthode B utilise le *langage des substitutions généralisées*, qui est une notation spécifique à la méthode. Dans d'autres techniques de spécification orientées modèle, comme Z ou VDM, les opérations sont décrites en utilisant des prédicats *avant-après*. Ceux-ci sont des prédicats logiques qui lient la valeur des variables d'état avant l'activation de l'opération, à la valeur des mêmes variables juste après l'activation de l'opération.

Comme nous allons le voir dans le paragraphe suivant, il existe une transformation systématique des substitutions généralisées en prédicats avant-après.

1.3 Langage des substitutions généralisées

Le langage des substitutions généralisées est la notation utilisée dans la méthode B pour la spécification des aspects dynamiques du système. Ce langage est constitué de substitutions de base issues d'une généralisation de la notion de substitution simple. Afin de faciliter l'utilisation du langage, un *sucre syntaxique* a été défini. Dans le paragraphe 1.3.1, nous expliquons la sémantique des substitutions généralisées. Dans le paragraphe 1.3.2 les substitutions de base et le sucre syntaxique sont décrits. Enfin, quelques propriétés définies pour les substitutions, notamment terminaison, faisabilité et transformation en prédicat avant-après sont décrites dans le paragraphe 1.3.3.

1.3.1 Sémantique des substitutions généralisées

D'une manière analogue à la technique introduite dans [Dijkstra 1976], les substitutions généralisées sont des transformateurs de prédicats. Le prédicat $[S]P$, où S est une substitution et P un prédicat, dénote la plus faible pré-condition pour que la substitution S établisse la post-condition P. L'*établissement* de la post-condition P signifie que toute exécution de S commençant dans un état satisfaisant $[S]P$ se *termine* dans un état *satisfaisant* P. La notion de *plus faible* pré-condition exprime le fait que toute pré-condition Q permettant d'établir P est telle que : $Q \Rightarrow [S]P$.

Par exemple, le prédicat $x > 4$ est la plus faible pré-condition pour que la substitution $x := x + 1$ établisse la post-condition $x > 5$:

$$[x := x + 1](x > 5) \Leftrightarrow x + 1 > 5 \Leftrightarrow x > 4$$

En particulier, l'exécution de $x := x + 1$ lorsque $x = 10$ se termine dans un état satisfaisant $x > 5$, et nous avons bien : $x = 10 \Rightarrow x > 4$.

1.3.2 Substitutions de base et sucre syntaxique

Le langage des substitutions généralisées est constitué d'un petit nombre de substitutions de base. Dans le Tableau 1.1, nous montrons une vue générale de ces substitutions avec leur sémantique décrite par la transformation de prédicats.

Avant d'expliquer plus en détail chacune de ces substitutions, mentionnons que le développeur B ne manipule pas directement le langage des substitutions généralisées. Dans les opérations des composants B, les substitutions de base sont généralement exprimées à l'aide d'un jeu d'instructions appelé le *sucre syntaxique*. Ces instructions permettent de se rapprocher d'un langage de programmation plus classique et offrent une meilleure lisibilité. Leur sémantique est déduite de leur traduction en substitutions

de base. Quelques exemples de ces instructions sont donnés dans le Tableau 1.2. Nous reprendrons ces exemples pour illustrer l'utilisation des substitutions de base correspondantes.

Tableau 1.1. Les substitutions généralisées de base : x, y et z sont des variables, E et F sont des expressions, R et P sont des prédicats, S et T sont des substitutions

Syntaxe	Sémantique
Substitution simple	$[x := E]R \Leftrightarrow$ remplacer toutes les occurrences libres de x par E dans R
skip	$[\mathsf{skip}]R \Leftrightarrow R$
Substitution multiple	$[x, y := E, F]R \Leftrightarrow [z := F][x := E][y := z]R$ $\quad z\backslash x, y, E, F, R$
Substitution pré-conditionnée	$[P \mid S]R \Leftrightarrow P \wedge [S]R$
Substitution gardée	$[P \Rightarrow S]R \Leftrightarrow P \Rightarrow [S]R$
Substitution de choix borné	$[S \,?\, T]R \Leftrightarrow [S]R \wedge [T]R$
Substitution de choix non borné	$[@z \cdot S]R \Leftrightarrow \forall z \cdot [S]R \quad z\backslash R$
Substitution de séquencement	$[S \,;\, T]R \Leftrightarrow [S][T]R$
Substitution d'itération	$[S\hat{}]R \Leftrightarrow [(S \,; S\hat{}) \,?\, \mathsf{skip}]R$

Tableau 1.2. Quelques exemples de sucre syntaxique

Sucre syntaxique	Définition
PRE P THEN S END	$P \mid S$
SELECT P THEN S END	$P \Rightarrow S$
CHOICE S OR T END	$S \,?\, P$
IF P THEN S ELSE T END	$(P \Rightarrow S) \,?\, (\neg P \Rightarrow T)$
VAR z IN S END	$@z \cdot S$
ANY z WHERE P THEN S END	$@z \cdot (P \Rightarrow S)$
$x : P$	ANY z WHERE $[x := z]P$ THEN $x := z$ END $\quad z\backslash P$
WHILE P DO S END	$(P \Rightarrow S)\hat{} \,;\, (\neg P \Rightarrow \mathsf{skip})$

Substitution simple. La substitution simple correspond à l'affectation usuelle d'une expression à une variable.

Substitution skip. La substitution skip représente la substitution qui *ne fait rien*, en d'autres termes qui n'a pas d'effet sur la pré-condition.

Substitution multiple. La substitution multiple $x, y := E, F$ correspond à l'affectation simultanée des expressions E et F à deux variables distinctes x et y. La sémantique de

cette substitution est définie en termes de substitutions simples (voir Tableau 1.1) : on introduit la variable *z* différente de *x* et de *y*, qui est non-libre dans *E*, *F* et *R*.

Cette composition multiple de substitutions *simples* est généralisée par la substitution parallèle, pour être appliquée à deux substitutions *généralisées*. La substitution parallèle $S \parallel T$, où les substitutions S et T modifient des variables distinctes, correspond à l'activation simultanée de celles-ci. Cette substitution peut toujours être éliminée par simple réécriture. En effet, nous avons les égalités suivantes :

$$(x := E \parallel y := F) = (x, y := E, F)$$
$$(S \parallel \mathsf{skip}) = S$$

Pour les autres substitutions généralisées, l'élimination de la substitution parallèle est définie par induction structurelle. Par exemple :

$$S \parallel (P \mid T) = P \mid (S \parallel T)$$

On se ramène ainsi à une substitution multiple.

Substitution pré-conditionnée. La substitution pré-conditionnée permet de spécifier des opérations qui ne peuvent être activées que sous certaines conditions exprimées par la pré-condition. Si ces conditions ne sont pas vérifiées, la substitution ne peut établir aucune post-condition. Ainsi, si P n'est pas vrai, $P \wedge [S]R$ n'est pas vrai quelle que soit la post-condition R. Une telle substitution est une substitution qui ne *se termine* pas. En pratique, la substitution pré-conditionnée est utilisée sous la forme du sucre syntaxique PRE *P* THEN *S* END.

Substitution gardée. La substitution gardée désigne une opération qui est activée sous une hypothèse P. Lorsque la garde n'est pas vérifiée, la substitution peut établir n'importe quelle post-condition . Ainsi, quand P n'est pas vrai, $P \Rightarrow [S]R$ est vrai quelle que soit la post-condition R. Une telle substitution est appelée *infaisable* ou *miraculeuse*. Le sucre syntaxique correspondant à la substitution gardée est : SELECT *P* THEN *S* END.

Substitution choix borné. La substitution choix borné exprime le choix entre deux substitutions. Ainsi, si une opération est spécifiée comme le choix entre deux substitutions, une future implémentation de cette opération peut choisir d'implémenter l'une ou l'autre de ces substitutions. Par conséquent, ceci introduit un indéterminisme borné dans la spécification.

Cette substitution est utilisée sous la forme du sucre syntaxique CHOICE *S* OR *T* END. Notons également la définition de la substitution conditionnelle IF (voir le Tableau 1.2) par le choix borné entre deux substitutions gardées dont les prédicats de garde sont mutuellement exclusifs.

Substitution choix non borné. La substitution choix non borné est une généralisation de la substitution choix borné. Soit une substitution S dépendant de la variable *z*. La

substitution $@z \cdot S$ représente toutes les substitutions S possibles indépendamment de la valeur de z. En d'autres termes, cette substitution propose à l'implémenteur de choisir n'importe quelle valeur de z pour implémenter S.

Cette substitution permet en pratique d'introduire une variable locale à la substitution S, comme l'illustre le sucre syntaxique VAR z IN S END (voir le Tableau 1.2). Le sucre syntaxique ANY z WHERE P THEN S END permet d'utiliser dans la substitution S une variable z satisfaisant P. Si plusieurs valeurs satisfont le prédicat P la substitution définit alors un comportement indéterministe. Dans la définition de $x : P$, la variable locale z sert à spécifier la valeur de x après activation de la substitution : ce sucre syntaxique permet donc d'exprimer directement une post-condition sur x.

Substitution de séquencement. La sémantique de la substitution de séquencement est similaire à celle de l'opérateur de séquencement « ; » dans les langages de programmation classiques. Cette substitution signifie que les deux substitutions composantes se suivent dans l'ordre.

Substitution d'itération. La dernière substitution de base que nous avons mentionnée dans le Tableau 1.1, est la substitution d'itération. La sémantique de cette substitution est formellement définie dans le B-Book en utilisant le concept de point fixe, et en faisant référence à un modèle ensembliste des substitutions généralisées que nous ne présentons pas ici. De manière informelle, la substitution d'itération $T^{\wedge}$ exprime le fait de répéter une substitution T. Cette substitution de base n'est utilisée que dans la définition de la boucle WHILE qui a la forme suivante (voir le Tableau 1.2) :

WHILE P DO S END

Cette instruction, comme le concept classique de boucle dans les langages de programmation, signifie que la substitution S est répétée tant que la condition P est vérifiée.

1.3.3 Terminaison, faisabilité et prédicat avant-après

Dans le paragraphe1.3.2, nous avons expliqué la sémantique de la substitution pré-conditionnée en utilisant la notion de terminaison et celle de la substitution gardée en utilisant la notion de faisabilité. Ces concepts sont formellement définis dans le B-Book pour toutes les substitutions généralisées.

Une substitution ne se termine pas si elle n'établit aucune post-condition. Pour une substitution S, $\mathsf{trm}(S)$ spécifie le prédicat qui est vrai si et seulement si la substitution S se termine. Il est défini par le prédicat :

$$\mathsf{trm}(S) \Leftrightarrow [S](x = x)$$

A partir de cette définition, la terminaison de toutes les substitutions généralisées peut être déduite. Par exemple, la terminaison de la substitution pré-conditionnée est le prédicat :

$$\begin{aligned} \mathsf{trm}\,(P \mid S) &\Leftrightarrow [P \mid S](x = x) \\ &\Leftrightarrow P \wedge [S](x = x) \\ &\Leftrightarrow P \wedge \mathsf{trm}\,(S) \end{aligned}$$

Ce prédicat signifie que la substitution pré-conditionnée $P \mid S$ se termine si et seulement si la pré-condition P est vraie et la substitution S se termine.

Une substitution est infaisable si elle établit n'importe quelle post-condition. Le prédicat $\mathsf{fis}\,(S)$ spécifie le prédicat qui est vrai si et seulement si la substitution S est faisable. Il est défini comme suit :

$$\mathsf{fis}\,(S) \Leftrightarrow \neg[S](x \neq x)$$

Par exemple, nous pouvons calculer le prédicat fis de la substitution gardée :

$$\begin{aligned} \mathsf{fis}\,(P \Rightarrow \quad S) &\Leftrightarrow \neg[\, P \Rightarrow \quad S](x \neq x) \\ &\Leftrightarrow \neg(P \Rightarrow [S](x \neq x)) \\ &\Leftrightarrow P \wedge \neg[S](x \neq x) \\ &\Leftrightarrow P \wedge \mathsf{fis}\,(S) \end{aligned}$$

Ce prédicat signifie que la substitution gardée $P \Rightarrow \quad S$ est faisable si et seulement si la garde P est vraie et la substitution S est faisable.

Il existe une correspondance systématique entre les substitutions généralisées et les prédicats avant-après. Le prédicat avant-après d'une substitution généralisée S, portant sur la variable x, est défini par le prédicat :

$$\mathsf{prd}_x\,(S) \Leftrightarrow \neg[S](x' \neq x)$$

Est utilisée la convention classique qui dénote la valeur après en primant la valeur avant de la variable. Ce prédicat exprime la liaison entre la valeur après de x et sa valeur avant : après l'activation de S, x et x' ne seront pas distincts.

Dans le calcul du prédicat avant-après, il est important de connaître la liste des variables sur lesquelles porte la substitution considérée. Avec les substitutions généralisées, il est inutile de préciser les variables qui ne sont pas modifiées alors qu'elles sont explicitement désignées avec les prédicats avant-après. Par exemple, $\mathsf{prd}_{x,y}\,(x := E)$ est le prédicat avant-après de la substitution $x := E$, qui porte sur les variables x et y mais ne modifie que la variable x. D'après la définition du prédicat avant-après, il est calculé de la manière suivante :

$$\begin{aligned} \mathsf{prd}_{x,y}\,(x := E) &\Leftrightarrow \neg[x := E](x' \neq x \vee y' \neq y) \\ &\Leftrightarrow \neg(\, x' \neq E \vee y' \neq y) \\ &\Leftrightarrow (x' = E \wedge y' = y) \end{aligned}$$

Enfin, nous mentionnons les deux propriétés suivantes qui ont été prouvées dans le B-Book :

1. La première propriété est : $\mathsf{fis}\,(S) \Leftrightarrow \exists x' \cdot \mathsf{prd}_x\,(S)$

Ceci indique que la substitution *S*, portant sur la variable *x*, est faisable si et seulement si une valeur après *x'* peut être calculée à partir de la valeur avant *x*.

2. La seconde propriété est l'existence d'une forme normale, sous laquelle toute substitution peut être mise :

$$S = \mathsf{trm}(S) \mid @x' \cdot (\mathsf{prd}_x(S) \Rightarrow \quad x := x')$$

Cette propriété montre que les prédicats trm et prd caractérisent entièrement toute substitution généralisée.

1.4 Concept de raffinement

Dans ce paragraphe, nous nous intéressons à la définition de la notion de raffinement dans le cadre de la méthode B. Le raffinement est une technique qui consiste à transformer le modèle abstrait d'un système en un modèle moins abstrait [Morgan 1994; Back et von Wright 1998]. En B, les machines abstraites sont graduellement raffinées jusqu'à arriver à des versions suffisamment concrètes pour pouvoir être automatiquement traduites dans un langage de programmation.

Le raffinement d'une machine abstraite conserve la même interface et le même comportement vis-à-vis de l'utilisateur extérieur que son abstraction, mais reformule les données et les opérations à l'aide d'éléments plus concrets. Le raffinement d'une machine abstraite se décline donc en termes de raffinement de sa partie statique et en termes de raffinement de sa partie dynamique. Avant d'expliquer le raffinement des machines abstraites, nous nous concentrons sur la définition du raffinement pour les substitutions généralisées.

1.4.1 Raffinement des substitutions généralisées

Chaque substitution raffinée doit réaliser ce qui est spécifié dans l'abstraction. Ainsi, la substitution S_1 est raffinée par S_2 (noté $S_1 \lozenge S_2$) si et seulement si toute post-condition *R* établie par S_1 l'est aussi par S_2 : $[S_1]R \Rightarrow [S_2]R$.

Il existe deux types de transformation d'une substitution qui constituent un raffinement de celle-ci.

Dans le premier cas, l'indéterminisme est réduit. Un exemple classique de ce type de raffinement est le raffinement de la substitution choix borné : la substitution *S* Í CHOICE S_1 OR S_2 END est raffinée par la substitution S_1. D'après la sémantique de cette substitution, pour toute post-condition *R,* si $[S]R$ est vérifié alors $[S_1]R$ est également vérifié. Cela signifie que si S_1 est activée dans un état où *S* se termine, elle se termine aussi et l'état final vérifie une post-condition au moins aussi forte.

Dans le second cas, la pré-condition est affaiblie. Par exemple la substitution S_1 Í PRE *P* THEN *S* END est raffinée par la substitution S_2 Í SELECT *P* THEN *S* END. Clairement $[S_1]R$ implique $[S_2]R$. Soulignons que dans ce cas le raffinement *fait*

plus que son abstraction, dans le sens où il peut se terminer dans les situations où son abstraction ne se terminerait pas. Le raffinement étant toujours activé dans les conditions de terminaison de son abstraction, l'utilisateur de cette substitution *ne se rendra pas compte* de cette différence.

D'après ce qui précède, et en rappelant que les prédicats trm et prd caractérisent entièrement toute substitution généralisée, la relation de raffinement peut être reformulée comme suit :

$$S_1 \lozenge S_2 \quad \Leftrightarrow \quad (\mathsf{trm}\ (S_1) \Rightarrow \mathsf{trm}\ (S_2)) \wedge (\mathsf{prd}_x\,(S_2) \Rightarrow \mathsf{prd}_x\,(S_1))$$

Dans cette définition, où S_1 et S_2 sont supposées porter sur la même liste de variables x, on retrouve explicitement l'affaiblissement de la pré-condition et la diminution de l'indéterminisme.

Notons que nous avons la propriété suivante : $S_1 \lozenge S_2 \wedge S_2 \lozenge S_1 \Leftrightarrow S_1 = S_2$. De plus, la relation de raffinement est transitive : $S_1 \lozenge S_2 \wedge S_2 \lozenge S_3 \Rightarrow S_1 \lozenge S_3$.

1.4.2 Raffinement des machines abstraites

Le raffinement des machines abstraites est défini dans le B-Book dans un premier temps à l'aide de *substitutions externes*, décrivant le comportement observable des machines abstraites. Une substitution externe est une séquence finie d'appels aux opérations de la machine abstraite, pouvant être implémentée sur la machine. Elle ne contient aucune référence aux variables d'état. Ceci correspond au principe d'encapsulation d'état : il n'est accessible qu'au travers des opérations.

Pour mieux expliquer la notion de substitution externe prenons l'exemple des deux machines suivantes, extrait du B-Book [Abrial 1996] :

MACHINE
Little_Example_1
VARIABLES
y
INVARIANT
$y \in \mathtt{F}(\mathbf{NAT_1})$
INITIALISATION
$y := \emptyset$
OPERATIONS
enter (n) ĺ
PRE $n \in \mathbf{NAT_1}$ **THEN**
$y := y \cup \{n\}$
END ;
$m \leftarrow$ **maximum** ĺ
PRE $y \neq \emptyset$ **THEN**
$m :=$ **max** (y)
END

MACHINE
Little_Example_2
VARIABLES
z
INVARIANT
$z \in$ **NAT**
INITIALISATION
$z := 0$
OPERATIONS
enter (n) ĺ
PRE $n \in \mathbf{NAT_1}$ **THEN**
$z :=$ **max** ($\{z, n\}$)
END ;
$m \leftarrow$ **maximum** ĺ
PRE $z \neq 0$ **THEN**
$m := z$
END

END	**END**

Les machines abstraites *Little_Example_1* et *Little_Example_2* portent respectivement sur les variables d'état $y \in \mathbb{F}(\mathrm{NAT}_1)$ et $z \in \mathrm{NAT}$ et ont deux opérations enter (n) et $m \leftarrow$ maximum. Elles ont la même signature (mêmes noms d'opérations et mêmes paramètres d'entrée et de sortie pour chaque opération), mais l'état et le corps des opérations sont exprimés différemment. La substitution *T* est un exemple de substitution externe pouvant être implémentée sur *Little_Example_1* et *Little_Example_2* :

T ΄ BEGIN
enter (*2*);
enter (*3*);
$x \leftarrow$ maximum
END

Pour implémenter la substitution externe *T* sur ces deux machines, on procède comme suit. Les machines sont d'abord initialisées, puis les appels aux opérations sont expansés par leur corps tel qu'il est défini dans chaque machine, les paramètres formels étant remplacés par les paramètres d'appel. Nous obtenons de cette manière deux substitutions que l'on appellera $T_{Little\_Example\_1}$ et $T_{Little\_Example\_2}$:

$T_{Little\_Example\_1}$ ΄ BEGIN
$y := \emptyset$;
PRE $2 \in \mathrm{NAT}_1$ THEN $y := y \cup \{2\}$ END ;
PRE $3 \in \mathrm{NAT}_1$ THEN $y := y \cup \{3\}$ END ;
PRE $y \neq \emptyset$ THEN $x := \max(y)$ END
END

$T_{Little\_Example\_2}$ ΄ BEGIN
$z := 0$;
PRE $2 \in \mathrm{NAT}_1$ THEN $z := \max(\{z, 2\})$ END ;
PRE $3 \in \mathrm{NAT}_1$ THEN $z := \max(\{z, 3\})$ END ;
PRE $z \neq 0$ THEN $x := z$ END
END

$T_{Little\_Example\_1}$ et $T_{Little\_Example\_2}$ ne portent pas sur le même ensemble de variables et ne peuvent donc être comparées. Pour ce faire, $T_{Little\_Example\_1}$ et $T_{Little\_Example\_2}$ sont généralisées, par la substitution choix non-borné, sur toutes les valeurs possibles des variables d'état des machines. On obtient alors respectivement :

$@y \cdot (y \in \mathbb{F}(\mathrm{NAT}_1) \Rightarrow \quad T_{Little\_Example\_1\_})$ et
$@z \cdot (z \in \mathrm{NAT} \Rightarrow \quad T_{Little\_Example\_2\_})$

qui portent sur la même variable *x*, récupérant la valeur de retour de l'opération maximum dans chaque cas. Dans cet exemple, les deux substitutions sont équivalentes à $x := 3$. Ainsi, en cachant les états internes, les machines

Little_Example_1 et *Little_Example_2* offrent des services identiques du point de vue de la substitution externe *T*.

Plus généralement, soient *M* et *N* deux machines abstraites ayant la même signature, mais encapsulant des variables d'état distinctes $v_M \in E_M$ et $v_N \in E_N$. *M* et *N* sont observationnellement équivalentes si et seulement si elles ne peuvent être distinguées par aucune substitution externe *T* :

$$@v_M \cdot (v_M \in E_M \Rightarrow \quad T_M) = @v_N \cdot (v_N \in E_N \Rightarrow \quad T_N) \qquad \text{pour tout } T$$

De la même façon, *M* est raffinée par *N* si et seulement si :

$$@v_M \cdot (v_M \in E_M \Rightarrow \quad T_M) \lozenge @v_N \cdot (v_N \in E_N \Rightarrow \quad T_N) \qquad \text{pour tout } T$$

Le raffinement des machines abstraites est ainsi défini en termes de raffinement de substitutions généralisées. Notons que cette définition est très générale et complexe du point de vue de la preuve d'un développement B, car elle implique une quantification universelle informelle sur toutes les substitutions externes. Il est démontré dans le B-Book qu'il est possible de la remplacer par des obligations de preuve concernant uniquement *M* et *N*, à condition d'avoir une relation totale entre l'état concret et l'état abstrait. Dans le paragraphe suivant, nous montrons comment le raffinement d'une machine abstraite est en pratique spécifié dans la méthode B.

1.4.3 Composants de type REFINEMENT et IMPLEMENTATION

Dans la méthode B, le raffinement d'une machine abstraite est spécifié dans un composant de type REFINEMENT. Un composant de type REFINEMENT n'est pas une machine abstraite à part entière. Il contient le supplément qui doit être ajouté à un composant MACHINE afin de construire la machine abstraite correspondante.

Dans la Figure 1.2, nous montrons comment une machine abstraite *M* et son raffinement peuvent être syntaxiquement combinés afin de construire une autre machine abstraite supposée raffiner *M*. Cette machine équivalente n'est jamais explicitement écrite, car il est possible de la produire systématiquement à partir de la machine abstraite *M* et son REFINEMENT. La relation totale entre l'état concret et l'état abstrait est définie par : $\{z,y \mid I \wedge J\}$. Son domaine est constitué de toutes les valeurs de *z* satisfaisant l'invariant de *M_N*. Les obligations de preuve du composant *N* (voir le paragraphe 1.6) constituent alors une condition suffisante pour que la machine *M_N* soit un raffinement de *M*.

La syntaxe des composants de type REFINEMENT est similaire à celle d'un composant de type MACHINE. Le mot clef MACHINE est remplacé par REFINEMENT et le nom du composant raffiné est écrit dans la clause REFINES. Un raffinement devant conserver la même interface vis-à-vis de l'utilisateur extérieur que son abstraction, le composant REFINEMENT ne modifie pas les paramètres formels de la machine, ni le nom et les paramètres d'entrée et de sortie des opérations. Comme expliqué plus haut, le raffinement de la partie statique de la machine abstraite est spécifié par le changement de l'espace d'état, et la relation totale liant cet état concret à l'état abstrait

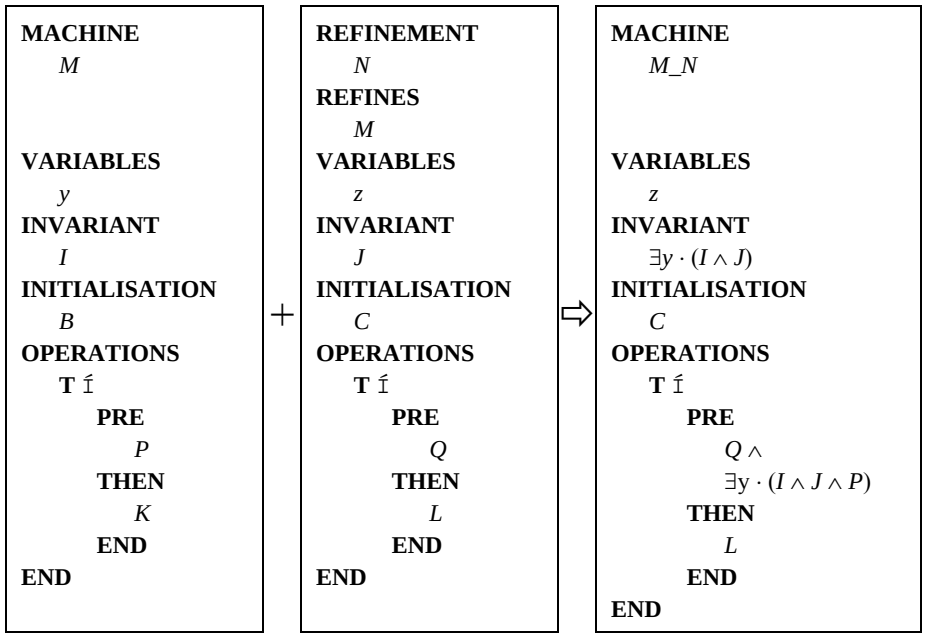


Figure 1.2. La machine abstraite équivalente construite à partir d'une machine abstraite et son raffinement

est déduite de l'invariant du composant REFINEMENT. Dans l'exemple du paragraphe précédent, l'invariant *J* d'un REFINEMENT permettant d'obtenir *Little_Example_2* à partir de *Little_Example_1* pourrait être : $z = \max(y \cup \{0\})$. Le composant REFINEMENT contient ensuite un ensemble d'opérations raffinant celles de l'abstraction. Notons que la combinaison syntaxique des composants MACHINE et REFINEMENT suppose que leurs variables d'état ont des noms distincts (voir notamment la quantification des variables abstraites dans l'invariant de M_N, et dans la pré-condition de l'opération). S'il existe, dans un REFINEMENT, une variable d'état de même nom que dans le composant raffiné, par exemple une variable *x*, on procède comme suit pour obtenir la machine équivalente : une des variables est renommée – par exemple la variable *x* du REFINEMENT est renommée en *x_0* – et l'invariant de la machine équivalente devient $\exists x \cdot (I \wedge J \wedge x = x\_0)$. Le prédicat $x = x\_0$, implicitement rajouté dans l'invariant concret, est un invariant *de liaison*.

Le raffinement peut être spécifié de façon incrémentale, par une succession de composants REFINEMENT. On pourrait ainsi compléter l'exemple de la Figure 1.2 par un composant *O*, contenant une clause « REFINES *N* », et permettant d'obtenir une machine équivalente *M_N_O* supposée raffiner *M_N* (et donc *M*, par transitivité). Le dernier raffinement d'un composant MACHINE est appelé une IMPLEMENTATION. Dans un composant IMPLEMENTATION les données et les opérations sont exprimées en utilisant des structures de données et des substitutions concrètes. Seul un sous-ensemble de la notation B peut être utilisé dans un composant IMPLEMENTATION.

Dans ce sous-ensemble, appelé B0, les opérations sont constituées de substitutions concrètes pouvant être directement traduites dans un langage de programmation. De même, les données utilisées doivent avoir un type implémentable : par exemple, intervalle d'entiers, tableau de booléens, etc.

Notons que les paramètres formels des opérations n'étant pas modifiés par le processus de raffinement, ils doivent, dès leur définition dans un composant MACHINE, avoir un type implémentable. De façon similaire, la méthode B offre la possibilité de déclarer, dans un composant abstrait, des variables d'état concrètes (clause CONCRETE_VARIABLES). Ces variables sont garanties exister dans tous les raffinements de ce composant, y compris l'implémentation finale, et doivent donc également avoir un type implémentable. Ces variables gardant le même nom, elles donnent lieu à des invariants de liaison.

1.4.4 Retour sur la notion de raffinement de machines abstraites

Une conséquence surprenante de la définition de raffinement en B est la possibilité de spécifier des machines abstraites, modélisant des *boîtes noires*, qui peuvent être raffinées par n'importe quelle autre boîte noire. Ceci est souligné dans le B-Book par l'exemple de la Figure 1.3 : en effet, il est possible de prouver que le composant *R* constitue un raffinement du composant *M*.

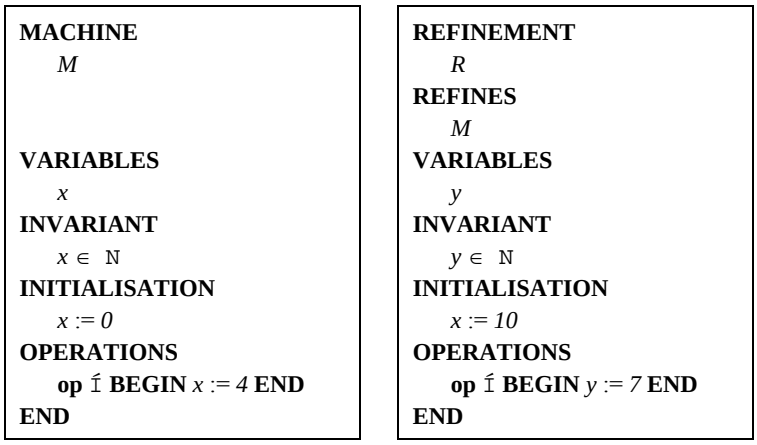


Figure 1.3. Exemple de raffinement de boîte noire

Ce résultat indésirable s'explique en faisant référence à la notion de substitution externe, présentée dans le paragraphe 1.4.2. L'opération ne retournant aucune valeur, le seul comportement observable par une substitution externe est la terminaison des séquences d'appel à cette opération. Les deux machines abstraites sont en fait observationnellement équivalentes (l'une raffine l'autre, et vice-versa).

Les obligations de preuve de *R*, qui constituent une condition suffisante (mais non nécessaire) pour que la machine équivalente soit un raffinement correct de *M*, ne permettent pas non plus d'éliminer cette anomalie. Le problème vient du fait qu'aucune relation explicite entre les variables de *M* et *R* n'a été spécifiée : par défaut, la relation totale entre états concret et abstrait est $\{y, x \mid x \in \mathtt{N} \wedge y \in \mathtt{N}\}$, c'est-à-dire que toutes les valeurs de *y* sont en relation avec toutes les valeurs de *x*. Les obligations de preuve de raffinement auraient éventuellement conduit à rejeter le composant R si une relation plus restrictive avait été spécifiée. En particulier, ce serait le cas si les variables d'état portaient le même nom *x* dans les composants abstrait et concret (et notamment si l'entier *x* avait été déclaré comme une variable concrète, devant être présente dans le raffinement) : les obligations de preuve (paragraphe 1.6) portant sur l'invariant de liaison n'auraient pu être déchargées.

1.5 Architecture de projets B

Le développement de logiciels en B s'effectue selon un processus incrémental. Ce processus repose sur deux mécanismes essentiels qui sont le mécanisme de raffinement (lien REFINES) et le mécanisme de décomposition en couches logicielles (lien IMPORTS). Il existe également d'autres mécanismes introduisant des liens entre composants (SEES, INCLUDES, USES). Ces différents mécanismes sont montrés dans la Figure 1.4 sur l'architecture symbolique d'un développement en B.

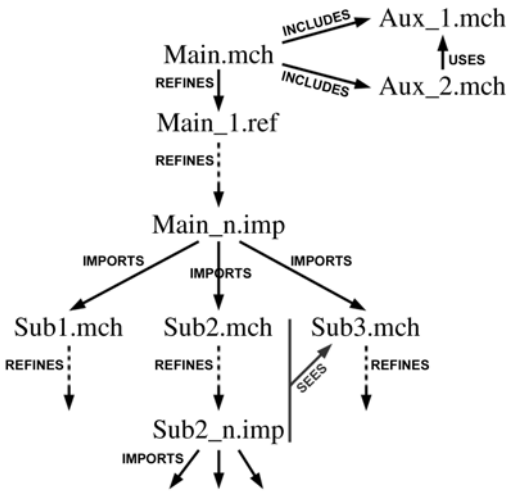


Figure 1.4. Structure générale d'un développement en B

En B, le mécanisme de raffinement n'est pas uniquement utilisé pour passer de la spécification au code, il est également utilisé comme une technique de spécification :

la spécification formelle d'une machine abstraite est raffinée à la fois pour obtenir des versions de plus en plus concrètes et pour rentrer les détails de la spécification informelle d'origine. Dès que la spécification devient trop complexe, elle est décomposée en sous-problèmes qui sont développés indépendamment. Cette décomposition est introduite par la clause IMPORTS. La tâche de raffinement est ainsi décomposée en tâches de raffinement de chacun des sous-systèmes importés. L'ossature d'un projet B est donc constituée d'une succession de liens REFINES et IMPORTS : un composant MACHINE est raffiné jusqu'à son IMPLEMENTATION ; cette IMPLEMENTATION utilise les services offerts par une ou plusieurs MACHINEs importées (elle appelle les opérations définies dans ces machines) ; les composants importés sont à leur tour raffinés, etc.

Le lien de raffinement a été présenté dans le paragraphe 1.4.3. Le lien d'importation définit également des règles de combinaison syntaxique des composants concernés. Nous ne montrons pas ici ces règles, elles seront données dans le chapitre 3. Mentionnons que l'importation crée une instance de chaque MACHINE importée. Si une MACHINE est paramétrée, la valeur des paramètres doit être spécifiée au moment de l'importation. Si plusieurs instances d'une même MACHINE coexistent dans une architecture B, elles doivent être importées sous des noms distincts. L'utilisation des services d'une MACHINE importée est indépendante de la représentation de l'état interne de celle-ci : dans les opérations d'un composant IMPLEMENTATION, il n'y a pas d'accès direct aux variables d'une MACHINE importée (exception faite des variables concrètes, qui peuvent être consultées directement). Ceci ouvre la possibilité d'utiliser les services d'un raffinement de la MACHINE importée, en lieu et place de celle-ci.

Le lien SEES permet d'établir des courts-circuits entre des branches indépendantes du développement. Un composant de type MACHINE, REFINEMENT ou IMPLEMENTATION peut voir une instance de composant de type MACHINE afin de consulter les constituants de celle-ci sans les modifier. Les règles de visibilité sont différentes dans chacun de ces trois cas. Dans le cas d'une IMPLEMENTATION, le lien SEES respecte l'encapsulation de l'état de l'instance de MACHINE vue, ceci afin de permettre le raffinement indépendant de celle-ci. Le partage de données introduit par le lien SEES est en effet conservé tout au long du développement, et sera présent dans l'implémentation finale du projet.

Contrairement aux liens IMPORTS et SEES, INCLUDES et USES ne sont pas conçus dans l'optique du raffinement des composants vers lesquels on établit un lien. Dans la Figure 1.4, les composants *Aux_1* et *Aux_2* ne sont destinés qu'à l'enrichissement local du texte formel de la MACHINE *Main*, selon certaines règles syntaxiques que nous ne détaillons pas ici.

1.6 Obligations de preuve

Dans un développement B, la preuve accompagne la construction du logiciel. Les obligations de preuve, générées systématiquement par la méthode, concernent la cohérence mathématique des composants de type MACHINE et la correction d'un raffinement vis-à-vis de ses versions plus abstraites.

Toutes les obligations de preuve contiennent en hypothèse des informations sur le contexte du composant. Ces informations concernent les propriétés des paramètres formels et celles des ensembles et constantes connus de ce composant. Nous notons par <*contexte*> le prédicat synthétisant ces informations. Les clauses de composition peuvent ajouter des informations dans le contexte selon les règles de visibilité.

La vérification de la cohérence mathématique d'un composant MACHINE consiste en deux types d'obligations de preuve :

1. Il faut prouver que l'initialisation établit l'invariant :

$$<\textit{contexte}> \Rightarrow [\textit{initialisation}]\ \textit{invariant} \qquad (1)$$

2. Il faut prouver que chaque opération conserve l'invariant. Si l'on suppose que les opérations ont la forme générique PRE *P* THEN *S* END, l'obligation de preuve suivante est générée pour chaque opération :

$$<\textit{contexte}> \wedge\ \textit{invariant} \wedge P \Rightarrow [S]\ \textit{invariant} \qquad (2)$$

La vérification de la correction d'un raffinement vis-à-vis de ses abstractions nécessite également deux types d'obligations de preuve. Pour le composant REFINEMENT ou IMPLEMENTATION correspondant au $n^{\text{ième}}$ raffinement, il s'agit de montrer que :

1. l'initialisation de ce composant établit l'invariant *sans contredire l'initialisation de l'abstraction* :

$$<\textit{contexte}> \Rightarrow [\textit{init}_n]\ (\neg[\textit{init}_{n\text{-}1}]\ \neg I_n) \qquad (3)$$

2. la pré-condition de l'opération du composant est vérifiée et l'opération du composant préserve l'invariant *sans contredire l'opération de l'abstraction* :

$$<\textit{contexte}> \wedge\ I \wedge I_1 \wedge \ldots \wedge I_n \wedge P \Rightarrow P_n \wedge [S_n]\ (\neg[S_{n\text{-}1}]\ \neg I_n) \qquad (4)$$

Dans les prédicats précédents, *I* est l'invariant du composant MACHINE et l'on suppose que les opérations ont la forme générique PRE *P* THEN *S* END. Nous notons par I_i, P_i et S_i respectivement l'invariant, la pré-condition de l'opération et l'action de l'opération du composant correspondant au $i^{ième}$ raffinement. Les substitutions définies aux niveaux *n-1* et *n* sont censées porter sur des variables distinctes. Tel n'est pas le cas lorsque les composants contiennent des variables concrètes, ou lorsque les opérations ont des paramètres de sortie. Dans les obligations de preuve (3) et (4), on doit alors renommer la variable d'état ou le paramètre de sortie commun. Ceci donne alors, par exemple :

$$<\textit{contexte}> \Rightarrow [[x := x\_0]\textit{init}_n]\ (\neg[\textit{init}_{n\text{-}1}]\ \neg(I_n \wedge x = x\_0)) \qquad (3')$$

La justification de ces obligations de preuve n'est pas immédiate. Il a été démontré dans le B-Book qu'elles constituent une condition suffisante pour garantir la correction d'un raffinement.

La méthode B ne génère pas d'obligations de preuve d'existence (par exemple, la preuve qu'il existe des valeurs de variables satisfaisant l'invariant). Ces preuves d'existence sont remplacées par des preuves de construction, qui sont beaucoup plus simples à établir. La preuve qu'il existe des valeurs satisfaisant l'invariant est indirectement introduite par les obligations de preuve (1) et (3). La preuve qu'il existe des valeurs d'entrée satisfaisant la pré-condition de l'opération est indirectement introduite lors de la preuve des composants qui appellent cette opération, instanciant ses paramètres d'entrée.

La preuve qu'il existe des valeurs de paramètres formels satisfaisant les contraintes d'une MACHINE est retardée au moment où la MACHINE est incluse ou importée dans un autre composant. Un composant incluant ou important une MACHINE paramétrée, doit instancier ses paramètres. L'obligation de preuve suivante est alors générée pour vérifier que ces valeurs satisfont les contraintes :

$$<\textit{contexte}> \Rightarrow [\textit{param\_formel} := \textit{param\_d\_appel}]\ (A \wedge C)$$

Le prédicat *A* exprime que les paramètres formels de type ensemble de la MACHINE importée ou incluse sont non-vides et finis, et le prédicat *C* est celui défini dans la clause CONSTRAINTS de la MACHINE importée ou incluse.

Les ensembles et les constantes d'une machine abstraite sont valués dans le composant IMPLEMENTATION. L'obligation de preuve suivante est alors générée pour vérifier que les valeurs choisies satisfont les propriétés :

$$<\textit{contexte'}> \Rightarrow [\textit{ensembles}, \textit{constantes} := \textit{valeurs}_1, \textit{valeurs}_2]\ (Pr \wedge \ldots \wedge Pr_n)$$

<*contexte'*> est un sous-ensemble de <*contexte*> concernant les valeurs $\textit{valeurs}_1$ et $\textit{valeurs}_2$, Pr_n est le prédicat défini dans la clause PROPERTIES de l'IMPLEMENTATION, Pr, …, Pr_{n-1} étant les prédicats définis dans la clause PROPERTIES de ses abstractions.

Notons finalement que le nombre de prédicats dans les obligations de preuve augmente avec le nombre d'étapes de raffinement et le nombre de liens de composition. Ainsi, les obligations de preuve deviennent de plus en plus complexes au fur et à mesure que les détails concrets sont introduits dans la spécification.

1.7 Outils commerciaux

Il existe deux ateliers logiciels couvrant la totalité du processus de développement en B. Il s'agit de l'Atelier B, commercialisé par STERIA-Méditerranée, et du B-Toolkit commercialisé par B-Core UK Ltd.

L'atelier de développement qui est le plus souvent utilisé en France est l'Atelier B. Il permet de gérer un ensemble de composants constituant un projet B. Les fonctionnalités principales concernent la vérification de la syntaxe et le contrôle de

type d'un composant, la génération des obligations de preuve, l'assistance à la preuve, et la traduction des IMPLEMENTATIONs en code exécutable.

L'outil de preuve permet de tenter la preuve automatique des obligations de preuve générées. Cet outil a plusieurs degrés de puissance, correspondant à l'application des différents ensembles de tactiques et de mécanismes de preuve. Plus la puissance est élevée, plus la preuve est longue et risque de boucler. Pour démontrer les obligations de preuves qui n'ont pas pu être automatiquement prouvées, un outil interactif de preuve est fourni.

D'autres fonctionnalités proposées par cet atelier sont :

- gestion de projet, un projet étant l'ensemble des composants B constituant le développement d'un logiciel ; cette fonctionnalité permet la création, l'ajout de composants, la maintenance des versions différentes d'un projet, etc.
- création automatique de documentation pour un projet ou pour un composant,
- génération d'un ensemble de fichiers hyper-textes permettant de parcourir un projet et de se déplacer à l'intérieur de ces constituants,
- animation des opérations d'un composant de type MACHINE. Cette fonctionnalité reste assez élémentaire et vise plutôt le prototypage des petites portions de la spécification afin d'aider le développeur à en comprendre le comportement.

1.8 Applications industrielles

La méthode B a été employée dans l'industrie ferroviaire française pour les applications logicielles dites critiques, pour lesquelles l'occurrence de défaillances pourrait avoir des conséquences catastrophiques (collision de deux trains, …). Ainsi, on peut citer les exemples suivants d'utilisation de la méthode B et de l'outil Atelier B [Behm et al. 1997]:

- le projet CTDC de développement des métros de Calcutta et la ligne 2 du métro du Caire,
- le projet SACEM simplifié d'extension de ligne A du RER de Paris,
- le projet KVB-SN de développement d'un sous-système utilisé par la SNCF pour les trains de grandes lignes,
- le projet Météor de développement de la ligne 14 du métro de Paris.

Le projet Météor est le système le plus complexe développé en utilisant la méthode B [Behm et al. 1999]. Il concerne le développement d'une ligne de métro totalement automatisée. Afin d'avoir un aperçu de la grandeur de ces projets, nous listons dans le Tableau 1.3 leur taille en nombre de lignes de spécification B, nombre de lignes de code source générées et nombre d'obligations de preuve. Le nombre de lignes de la spécification B est supérieur au nombre de lignes du code source, car il concerne toute l'architecture B, des composants MACHINE aux composants IMPLEMENTATION.

Tableau 1.3. Grandeur des projets développés en B en nombre de lignes de spécification B, nombre de lignes de code source et nombre d'obligations de preuve (OP)

Projet	No. de lignes B	No. de lignes de code	No. d'OP
SACEM simplifié	3500	2500 (Modula 2)	550
CTDC	5000	3000 (Ada)	700
KVB-SN	9000	6000 (Ada)	2750
Météor	115000	86000 (Ada)	27800

1.9 Conclusion

Ce chapitre nous a permis de présenter la méthode B, utilisée pour le développement formel de logiciels.

Un développement B est conçu d'une manière incrémentale et suit une architecture en couches. Les aspects statiques du comportement d'un système sont modélisés en utilisant les concepts mathématiques de la théorie des ensembles et la logique du premier ordre, et les aspects dynamiques en utilisant le langage des substitutions généralisées. Des obligations de preuve sont générées systématiquement par la méthode. Elles accompagnent le développement et garantissent la cohérence mathématique des composants ainsi que la correction du raffinement.

La méthode B est fondée sur des bases mathématiques anciennes, et bien établies. Des exemples d'utilisation de B dans un contexte industriel témoignent de la maturité de cette méthode.

Divers travaux de recherche se sont greffés autour de la méthode. On peut citer par exemple l'extension du champ d'application de B aux logiciels événementiels [Abrial et Mussat 1998], l'étude de la sémantique des liens de composition définis dans la méthode [Bert et al. 1996; Potet et Rouzaud 1998; Rouzaud 1999], la définition de métriques applicables à des projets B [Mariano 1997], ou encore la proposition d'une démarche méthodologique pour la spécification de systèmes critiques en B [Taouil-Traverson 1997].

Un des axes de recherche autour de la méthode B, motivé largement par ses applications industrielles, concerne l'amélioration du processus de vérification des logiciels développés en B. Nos travaux se situent dans ce cadre. Ils concernent l'utilisation d'une technique de vérification dynamique, le test, en complément aux preuves prescrites par la méthode B. Le chapitre suivant justifie l'intérêt de cette problématique, et définit les orientations prises par nos travaux.

Chapitre 2

Test et développement formel

2.1 Introduction

L'objectif de la vérification du logiciel est de révéler les fautes de conception, qui ont pu être introduites au cours de n'importe quelle phase du cycle de développement. La vérification doit alors accompagner le processus de développement afin de révéler les fautes, si possible, dès leur apparition [Laprie et al. 1996]. Ceci a comme objectif de diminuer le coût de leur élimination.

Dans le but d'une meilleur efficacité, différentes techniques de vérification sont généralement mises en œuvre. L'une de ces techniques est la preuve de programmes dont les fondements théoriques remontent à [Floyd 1967; Hoare 1969; Dijkstra 1976] et qui a donné lieu à divers travaux [Hoare 1978; Apt 1981; Apt 1984; Loeckx et Sieber 1984; Backhouse 1989]. Les notations formelles, dont la sémantique est basée sur les logiques mathématiques, permettent l'utilisation de cette technique, comme nous l'avons vu précédemment dans le cadre de la méthode B. Ce chapitre a pour but de positionner la place du test dans un processus de développement formel et de justifier l'intérêt des travaux menés sur ce thème pour un développement B.

La bibliographie sur les méthodes de test de programme est très vaste [Roper 1992]. Nous introduisons, dans le paragraphe 2.2, une classification générale de ces méthodes. Les paragraphes 2.3 et 2.4 présentent plus en détail quelques méthodes de test structurel et fonctionnel.

Le paragraphe 2.5 discute de la complémentarité du test et de la preuve dans le cadre d'un développement formel en général, puis dans le cadre de la méthode B en particulier.

Cette discussion nous conduit à envisager l'utilisation du test tôt dans le processus de développement, sur les modèles formels. Ceci nécessite cependant de pouvoir exécuter la spécification. Dans le paragraphe 2.6, nous présentons des techniques et des outils d'animation des spécifications orientées modèles Z, VDM et B.

2.2 Classification des techniques de test

Le test est une technique de vérification dynamique qui consiste à exécuter le programme en lui fournissant des entrées valuées, appelées les entrées de test, et à observer les sorties générées par le programme pour déterminer leur correction [Laprie et al. 1996].

Décider de l'exactitude des résultats fournis par le programme en réponse aux entrées de test, est connu sous le nom de *problème de l'oracle* [Weyuker 1982]. Il n'existe pas de solution générale à ce problème et il est souvent supposé que le testeur est capable de déterminer la correction des résultats dans un laps de temps raisonnable. Cependant, dans la littérature, quelques solutions automatiques ont été proposées. On peut citer par exemple le *test dos à dos* et l'utilisation des spécifications exécutables. Dans le premier cas, les résultats produits par le programme sont comparés avec ceux obtenus d'une deuxième version de celui-ci, développée indépendamment à partir du même dossier de spécification. Dans le second cas, les résultats sont comparés avec ceux obtenus de la spécification.

Le test exhaustif sur l'ensemble du domaine d'entrée étant impossible, le testeur est amené à choisir d'une manière pertinente un sous-ensemble de ce domaine. Les méthodes de détermination des entrées de test peuvent être classées selon deux aspects : le critère de sélection et la méthode de génération. La combinaison de ces deux aspects classifie les techniques de test en quatre grandes catégories qui sont représentées dans le Tableau 2.1.

Tableau 2.1. Classification des techniques de test selon le critère de sélection et la méthode de génération

	Critères structurels	Critères fonctionnels
Génération déterministe	Test déterministe structurel	Test déterministe fonctionnel
Génération probabiliste	Test statistique structurel	Test statistique fonctionnel

Les critères de sélection spécifient, à partir d'un modèle du logiciel cible, un ensemble d'éléments qui doivent être activés (couverts) au cours du test. Le modèle utilisé peut être basé sur la structure interne du logiciel, et dans ce cas le test est appelé *structurel*. Il peut également être basé sur les fonctions attendues du logiciel, et le test est alors appelé *fonctionnel*.

Les critères structurels ont une vision *boîte de verre* du programme. Le modèle utilisé par ces critères est le graphe de contrôle. Le graphe de contrôle, construit à partir du code source, fournit une vue compacte de l'algorithme du programme. Les critères structurels sélectionnent un sous-ensemble de chemins sur ce graphe, un chemin étant un parcours complet entre le nœud d'entrée et un nœud de sortie. Ils sont utilisés préférentiellement pour les composants de petite taille, car le graphe de contrôle devient vite très complexe au fur et à mesure que la taille du code source croît.

Les critères fonctionnels ont une vision *boîte noire* du programme. Le test fonctionnel repose sur un modèle décrivant le comportement attendu du programme, mais contrairement à la technique structurelle, il ne dispose pas de modèle *standard*. Les modèles les plus connus sont les classes d'équivalence, les tables de décision et les automates à états finis. Pour les classes d'équivalence [Myers 1979], le critère de test consiste à choisir des valeurs d'entrée pour couvrir les classes spécifiées et leurs frontières. Pour une table de décision [Beizer 1990], le critère principal est la couverture de chacune des règles. Pour une machine à états finis [Ural 1992], les critères courants sont la couverture des états, des transitions, ou des séquences de transitions. Les spécifications formelles fournissent également un modèle adéquat pour la sélection des entrées de test. Dans [Raynaud et Saint-Gealme 1990; Bernot et al. 1991; Gaudel 1995b] par exemple, les entrées de tests sont générés à partir de spécifications algébriques, le critère de sélection étant la couverture des axiomes de la spécification, qui expriment les propriétés attendues du système.

Une fois qu'un critère de sélection a été retenu, les valeurs d'entrée doivent être générées de manière à activer tous les éléments spécifiés par le critère. Quel que soit le critère choisi, la génération des entrées de test peut être conduite de deux manières : la génération *déterministe* et la génération *statistique*.

La génération déterministe consiste en un choix sélectif de données qui activeront chaque élément au moins une fois. Le choix des données est généralement fait manuellement par le testeur, mais il existe aussi des outils automatiques de génération d'entrées de test. Dans [Dick et Faivre 1992] et [Marre 1995] par exemple, les entrées de test déterministe sont générées automatiquement à partir de spécifications formelles. Afin de réduire l'effort de test, on cherche généralement à minimiser le nombre d'entrées. Les entrées de test sont alors sélectionnées de manière à ce que chaque élément soit activé seulement une fois.

Les critères de test sont un moyen imparfait de pallier l'absence de modèle de faute de conception pour le logiciel. Activer les éléments spécifiés par ces critères une fois, ou un petit nombre de fois, ne semble pas suffisant pour révéler les fautes. Ceci constitue une limitation importante de l'approche déterministe.

L'approche test statistique [Waeselynck 1993; Thévenod-Fosse et al. 1995], vise à compenser l'imperfection des critères de sélection en générant des entrées de test de manière à ce que chaque élément soit activé plusieurs fois. Pour cela, un tirage aléatoire est réalisé, non pas en aveugle (à la différence du test aléatoire selon un distribution uniforme [Duran et Ntafos 1984]) mais guidé par une évaluation de la distribution des probabilités d'entrée à appliquer. Cette distribution est liée à la

couverture des éléments spécifiés par le critère de sélection et détermine la longueur du test pour atteindre une qualité de test désirée. Des résultats expérimentaux ont montré l'efficacité du test statistique [Thévenod-Fosse et Waeselynck 1998]. Par essence, ce type d'approche probabiliste repose sur la génération automatique d'un nombre élevé d'entrées de test, et nécessite donc une procédure d'oracle également automatisée.

2.3 Test structurel de programmes

Les critères définis à partir de l'analyse structurelle de programmes sont nombreux [Beizer 1990]. Ils spécifient un ensemble de chemins sur le graphe de contrôle qui doivent être activés au cours du test. Par activation de chemin, on entend la sélection des entrées de test de manière à ce que l'exécution du programme avec ces entrées corresponde au parcours de ce chemin sur le graphe. On peut distinguer deux grandes familles parmi les critères structurels : ceux basés sur le graphe de contrôle uniquement, et ceux prenant en compte le graphe de contrôle et le flot de données dans ce graphe. Dans les paragraphes suivants, nous présentons chacune de ces familles.

2.3.1 Critères basés sur le flot de contrôle

Le graphe de contrôle permet d'obtenir une vision synthétique de la structure du programme. Il contient un nœud d'entrée et éventuellement plusieurs nœuds de sortie. Les nœuds de ce graphe sont des blocs d'instructions qui sont exécutées dans l'ordre, et les arcs entre les nœuds correspondent aux branchements dans le programme. La Figure 2.1 montre un exemple de graphe de contrôle construit pour une implémentation de la fonction FACTORIELLE, *fac* = *n*!, où *n* est un entier naturel.

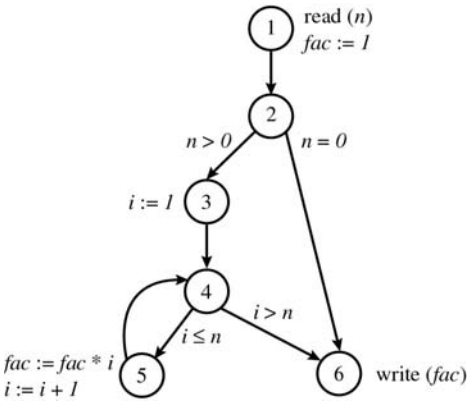


Figure 2.1. Exemple : graphe de contrôle de la fonction FACTORIELLE

Des exemples classiques de critères basés sur le flot de contrôle sont « toutes les instructions » et « toutes les branches », qui demandent respectivement que toutes les instructions (ou de façon équivalente tous les nœuds) et toutes les branches du graphe de contrôle soient activées au moins une fois pendant le test. Dans l'exemple de la Figure 2.1, une exécution avec une valeur positive de *n* est suffisante pour couvrir le critère « toutes les instructions ». Pour couvrir le critère « toutes les branches », deux exécutions au moins sont nécessaires : une avec $n > 0$ et l'autre avec $n = 0$.

Le critère structurel considéré comme étant le plus exhaustif est le critère « tous les chemins », qui demande l'activation de tous les chemins *exécutables* du graphe de contrôle. Un chemin est exécutable ou faisable, s'il existe une attribution de valeurs aux variables d'entrée qui activera ce chemin [Frankl et Weyuker 1986; Frankl et Weyuker 1988]. Par exemple, le chemin $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6)$ sur le graphe de la Figure 2.1 n'est exécutable pour aucune valeur de *n*.

Dès lors qu'un programme contient une boucle, il peut comporter un très grand nombre (voire même un nombre infini) de chemins. Par conséquent, le critère « tous les chemins » est souvent difficile ou impossible à couvrir. D'autres critères structurels, qui limitent le nombre d'itérations des boucles, ont été définis. Ces critères choisissent un sous-ensemble de chemins qui traversent les boucles un nombre borné de fois [Ntafos 1988].

2.3.2 Critères basés sur le flot de données

Ces critères visent à révéler les anomalies liées à l'état des données et à leur manipulation dans le programme [Rapps et Weyuker 1985]. Le modèle utilisé par ces critères est aussi le graphe de contrôle enrichi par l'introduction d'informations relatives à la définition et l'utilisation des variables. Les critères de flot de données proposent alors de couvrir des sous-chemins entre les nœuds du graphe de contrôle contenant les définitions des variables, et les nœuds (ou les arcs) référençant ces mêmes variables dans des calculs (ou des prédicats). On peut citer, par exemple, le critère *toutes les utilisations* ou ses versions plus restreintes *toutes les définitions*, *toutes les C-utilisations* et *toutes les P-utilisations*.

2.3.3 Relation d'inclusion

Il existe une notion formalisée de hiérarchie et d'ordonnancement partiel entre les critères de couverture structurelle des programmes. Ces critères sont classés selon la relation d'*inclusion* [Rapps et Weyuker 1985; Ntafos 1988], qui se base sur une comparaison des ensembles d'entrées de test. Un ensemble d'entrées de test *satisfait* ou couvre un critère, si ces entrées activent tous les éléments spécifiés par ce critère. Le critère C_1 inclut le critère C_2 si, quel que soit le programme considéré, tout ensemble d'entrées de test qui satisfait C_1 satisfait également C_2. Deux critères sont *incomparables* si aucun des deux n'est inclus par l'autre.

Un critère qui inclut un autre critère est plus sévère dans le sens où il exige la sélection d'un plus grand nombre d'entrées de test. Cette comparaison entre les critères structurels est possible car ils se basent sur le même modèle de logiciel. Les critères basés sur le flot de données sont tous inclus par le critère « tous les chemins », et ils incluent pour la plupart les critères « toutes les instructions » et « toutes les branches ». Même dans ce cadre, certains critères sont incomparables parce qu'ils considèrent des éléments structurels dissemblables. On peut citer comme exemple, les critères « toutes les instructions » et « toutes les définitions ».

2.4 Test fonctionnel à partir des spécifications orientées modèle

Les spécifications formelles fournissent un modèle qui peut être utilisé pour construire des jeux de test fonctionnel. Les approches de test diffèrent selon le formalisme utilisé. La méthode B appartenant à la famille des spécifications orientées modèle, nous nous concentrons sur les travaux qui portent sur ce type de spécifications. Dans le paragraphe 2.4.1, nous faisons la synthèse des critères de sélection d'entrées de test proposés dans la littérature. D'autres travaux, plus amont, se sont intéressés à la formalisation de notions théoriques relatives au test. Ces travaux font l'objet du paragraphe 2.4.2.

2.4.1 Critères de sélection d'entrées de test

Comme nous l'avons vu dans le chapitre précédent, dans les approches orientées modèle, la spécification du système est constituée des états que le système peut prendre, des relations d'invariance qui sont maintenues lorsque le système passe d'un état à un autre, et des opérations permettant de fournir un service et de modifier l'état. Les critères de sélection indiquent comment exploiter ces informations pour la conception de tests fonctionnels.

Le comportement dynamique spécifié dans les opérations est fortement dépendant de l'état interne. Or, cet état n'est généralement pas commandable ni observable au cours du test. L'activation des cas extraits de l'analyse de la spécification, et l'observation des réponses du système testé, ne peuvent s'effectuer que par des séquences d'appels aux opérations. Les approches de test proposées dans la littérature comportent donc deux aspects :

- analyse des opérations, prises individuellement ; cette analyse détermine un ensemble de cas à couvrir pour chaque opération ; les cas dépendent à la fois des entrées des opérations, et de l'état encapsulé ;
- génération de séquences d'appels aux opérations ; chaque opération dans la séquence amène le système dans un état à partir duquel l'opération suivante peut être activée.

Ces deux aspects sont présentés ci-après.

2.4.1.1 Couverture des opérations

Nous présentons, dans ce paragraphe, les approches définies pour déterminer les cas à couvrir pour chaque opération d'une spécification orientée modèle. Certaines de ces approches portent sur la notation Z [Hörcher et Peleska 1994; Hörcher 1995; Hörcher et Peleska 1995; Hierons 1997], et d'autres sur la notation VDM [Dick et Faivre 1992; Dick et Faivre 1993], mais elles sont basées sur des principes similaires.

On effectue une *analyse de partition*, qui décompose le domaine d'entrée de chaque opération en sous-domaines, pour lesquels au moins une valeur d'entrée devra être sélectionnée. En pratique, les sous-domaines peuvent ou non être disjoints. Les approches d'analyse de partition procèdent toutes à partir de la structure propositionnelle (disjonction $\vee$, conjonction $\wedge$, négation $\neg$) du prédicat avant-après. Chaque sous-domaine ainsi obtenu est appelé un *cas de test*, qui dépend à la fois des paramètres d'entrée de l'opération et de l'état avant. En préliminaire à l'analyse de partition, les constructions syntaxiques de la spécification sont mises à plat, les fonctions et les définitions récursives étant dépliées un nombre borné de fois. Ce traitement de la récursivité est analogue au traitement des boucles dans le test structurel de programmes.

L'approche d'analyse de partition proposée dans [Hörcher et Peleska 1994; Hörcher et Peleska 1995] utilise une réécriture en *forme normale disjonctive*[1]. Le prédicat avant-après *OP* spécifiant une opération est donc réécrit sous une forme $\vee_i OP_i$. Comme cela est schématisé par la Figure 2.2 l'intersection des sous-prédicats ainsi obtenus peut être non vide.

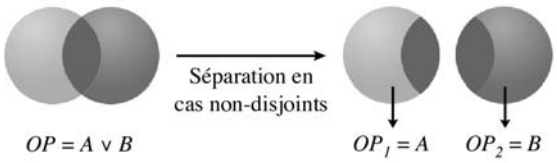


Figure 2.2. Sous-prédicats générés par l'approche de [Hörcher et Peleska 1995]

Chaque sous-prédicat OP_i porte sur les paramètres d'entrée et de sortie de l'opération, ainsi que sur l'état interne avant et après. Pour extraire des cas de test, qui décomposent le domaine d'entrée uniquement, une quantification existentielle sur les paramètres de sortie et l'état après doit être utilisée. Ainsi, en supposant que *var_sortie* désigne l'ensemble des variables de sortie de l'opération, et x' la valeur après de l'état x, le cas de test correspondant à chaque OP_i est obtenu par : $\exists var\_sortie, x' \cdot OP_i$.

L'approche présentée dans [Dick et Faivre 1992; Dick et Faivre 1993] consiste à décomposer le prédicat avant-après en sous-prédicats mutuellement exclusifs. Ceci se

[1] Un prédicat s'écrivant comme une disjonction de conjonctions de sous-prédicats atomiques est en forme normale disjonctive.

fait par un traitement de l'opérateur de disjonction logique $\vee$. Le prédicat $A \vee B$ est séparé en trois sous-prédicats : $A \wedge B$, $\neg A \wedge B$ et $A \wedge \neg B$. Le prédicat $A \Rightarrow B$ est également séparé en deux sous-prédicats : $\neg A$ et $A \wedge B$. Comme pour [Hörcher et Peleska 1995] le prédicat avant-après est ainsi réécrit sous une forme normale disjonctive, mais maintenant les sous-prédicats OP_i sont mutuellement exclusifs. Cette approche d'analyse de partition est illustrée par la Figure 2.3. De même que [Hörcher et Peleska 1995], pour générer des cas de test à partir de chaque OP_i, une quantification existentielle sur les paramètres de sortie et l'état après est utilisée.

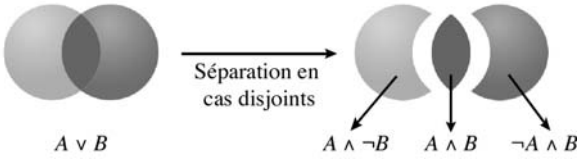


Figure 2.3. Sous-prédicats générés par l'approche de [Dick et Faivre 1993]

La décomposition proposée par [Dick et Faivre 1993] est plus fine que celle proposée par [Hörcher et Peleska 1995], augmentant considérablement le nombre de cas générés. Ainsi, cette décomposition pose des problèmes d'explosion combinatoire. Même pour une spécification relativement simple, un nombre important de cas de test est généré.

L'approche proposée par [Hierons 1997] réécrit le prédicat avant-après sous la forme $\vee_i(P_i \wedge Q_i)$, tel que :

- P_i, appelé un prédicat d'entrée, porte uniquement sur les paramètres d'entrée de l'opération et l'état avant,
- Q_i, appelé un prédicat de sortie, définit les paramètres de sortie de l'opération et l'état après.

La réécriture se fait à l'aide de trois règles définies dans [Hierons 1993]. Les P_i sont considérés comme des pré-conditions que l'on va chercher à décomposer, et les Q_i expriment le comportement de l'opération vis-à-vis de ces pré-conditions. Le fait de partitionner uniquement les P_i et non pas les $P_i \wedge Q_i$ est une approximation qui suppose que les Q_i n'affectent pas la pré-condition. Il faut mentionner que le prédicat $\vee_i(P_i \wedge Q_i)$, n'est pas sous forme normale disjonctive car P_i et Q_i peuvent contenir des disjonctions.

Après la réécriture, on n'a pas encore obtenu une partition du domaine d'entrée : les sous-domaines exprimés par les P_i peuvent se chevaucher. Des sous-domaines disjoints sont générés par traitement du prédicat $\vee_i P_i$. Ce traitement est le même que celui proposé par [Dick et Faivre 1993], mais ne porte que sur l'opérateur de disjonction le plus externe. Par exemple, le prédicat $(P_1 \wedge Q_1) \vee (P_2 \wedge Q_2)$ génère la partition suivante, qui détermine trois cas de test :

$$P_1 \wedge P_2 \qquad \neg P_1 \wedge P_2 \qquad P_1 \wedge \neg P_2$$

Cette partition est moins fine que celle de [Dick et Faivre 1993] puisque les P_i n'ont pas été décomposés, et les Q_i n'ont pas été pris en compte. Cependant, cette approche pose également des problèmes d'explosion combinatoire. Comme l'auteur le signale, une spécification initiale $\bigwedge_i (P_i \vee Q_i)$, $1 \leq i \leq n$, sera réécrite comme la disjonction de 2^n prédicats.

Pour illustrer les différentes décompositions proposées par chacune de ces trois approches, nous utilisons l'exemple très simple d'une opération spécifiée par le prédicat avant-après $(a \vee b) \wedge C$ tel que a et b sont des prédicats d'entrée et C est un prédicat de sortie. Ce prédicat a la forme désirée $P \wedge Q$ dans [Hierons 1997] et n'a pas besoin d'être réécrit. Les sous-prédicats générés par [Dick et Faivre 1993] et [Hörcher et Peleska 1995] sont montrés dans la Figure 2.4.

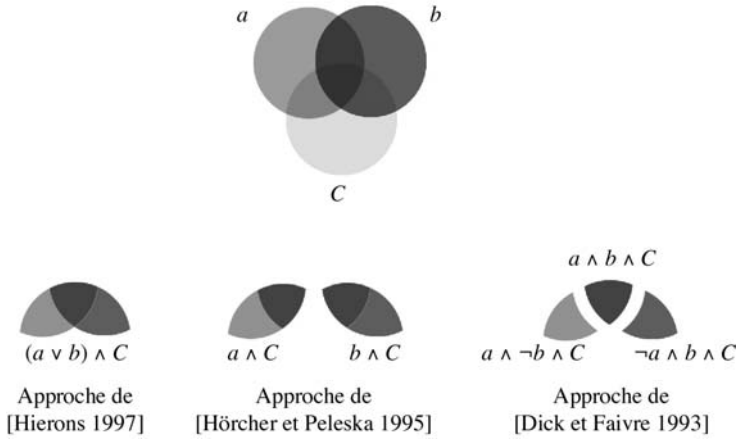


Figure 2.4. Comparaison des approches de génération de cas de test à l'aide d'un exemple

Toutes ces approches peuvent être raffinées en considérant les valeurs limites et en décomposant les opérateurs de la théorie des ensembles. Par exemple, $x \leq m$ peut être séparé en deux cas :

1) $x = m$ 2) $x < m$.

Le prédicat $e \in S_1 \cup S_2$ peut être séparé en trois cas :

1) $e \in S_1 \cap S_2$ 2) $e \in S_1 - S_2$ 3) $e \in S_2 - S_1$.

2.4.1.2 Génération des séquences d'appels aux opérations

Pour séquencer les appels aux opérations, [Dick et Faivre 1993] et [Hierons 1997] proposent de construire un automate à états finis à partir de la spécification, et de le tester en utilisant les critères classiques de couverture des automates [Ural 1992]. Les états de l'automate sont obtenus à partir de la partition du domaine d'entrée des opérations, et les transitions sont les opérations appelées dans un de leurs sous-domaines.

Nous avons vu dans le paragraphe précédent comment [Dick et Faivre 1993] décomposent le prédicat avant-après *OP* de chaque opération en sous-prédicats OP_i. Deux ensembles de contraintes sont extraits de chaque sous-prédicat, le premier décrivant l'état avant et le second décrivant l'état après. Cette extraction se fait par une quantification existentielle sur les variables qui ne concernent pas l'état voulu. Par exemple dans le cas de l'état avant, on quantifie sur les variables d'entrée et de sortie, et les variables d'état après. La disjonction de ces deux ensembles de contraintes est ensuite séparée en cas mutuellement exclusifs en utilisant les règles de séparation précédemment mentionnées (voir Figure 2.3). On obtient ainsi une partition de l'espace d'état du système.

Dans [Hierons 1997], les cas de test sont obtenus à partir de prédicats P_i portant uniquement sur les paramètres d'entrée et l'état avant des opérations. Pour générer les états de l'automate, on élimine les paramètres d'entrée de chaque cas de test, par quantification existentielle sur les variables correspondantes. Comme nous l'avons déjà mentionné pour les cas de test, les états générés de cette manière sont moins fins que ceux générés par l'approche de [Dick et Faivre 1993], et ce d'autant plus qu'on ne considère pas les états après des opérations.

Pour générer les transitions de l'automate, il faut trouver les opérations qui peuvent amener le système d'un état à l'autre. L'approche proposée par les auteurs consiste à évaluer les prédicats avant-après correspondant aux cas de test pour chaque opération. Soit le prédicat OP_i décrivant le comportement de l'opération *OP* pour le sous-domaine d'entrée correspondant au cas de test *i*. OP_i est supposé porter sur un paramètre d'entrée *var_entrée*, un paramètre de sortie *var_sortie*, et une variable d'état *x* apparaissant sous sa forme avant et après. Soient deux états *j* et *k* de l'automate, caractérisés respectivement par les prédicats S_j et S_k. Il existe une transition étiquetée par OP_i entre les états *j* et *k* si et seulement si il existe une valuation des variables *var_entrée*, *var_sortie*, *x* et *x'* telle que le prédicat $S_j \wedge OP_i \wedge [x := x'] S_k$ s'évalue à *true*.

La génération de l'automate, et particulièrement de ses transitions, constitue le point dur de ces approches. Aucun des travaux mentionnés ne dispose d'un outil opérationnel pour automatiser ce traitement.

2.4.1.3 Outil B-CASTING

Les approches que nous avons présentées dans les paragraphes précédents procèdent à partir de la structure propositionnelle de spécifications Z et VDM. Cependant, le même principe d'approche peut être appliqué dans le cadre de la méthode B. Une première illustration de ce type de traitement est présentée dans la méthode CASTING [Van Aertryck et al. 1997a; Van Aertryck et al. 1997b; Van Aertryck 1998].

CASTING est une méthode d'aide à la génération de séquences de test pour les logiciels. Cette méthode a pour but d'être générique pour pouvoir s'appliquer à différents formalismes. Le prototype d'un outil implémentant cette méthode a été développé et a été instancié pour accepter les spécifications B en entrée (B-

CASTING). B-CASTING traite uniquement les composants de type MACHINE sans clause de composition.

Pour instancier la méthode à un formalisme donné, la syntaxe de celui-ci doit être décrite à l'aide d'une grammaire attribuée. Deux attributs sont attachés aux symboles du formalisme : ces deux attributs vont être utilisés pour le calcul des cas de test associés à chaque opération. Le premier attribut est appelé *tr* pour *tr*aduction. Cet attribut traduit les structures du formalisme traité en prédicats avant-après. Les prédicats sont exprimés dans un langage spécifique à CASTING, pouvant être associé à un outil de résolution de contraintes. Le deuxième attribut est *xt* pour e*xt*raction. Cet attribut est défini pour permettre de raffiner les cas de test en décomposant les constructions syntaxiques du formalisme.

Pour chaque opération *op* du texte d'entrée, le calcul des attributs *tr* et *xt* produit la formule *F* suivante :

$$F = op.tr \wedge op.xt \wedge \{id' == id \mid id : \text{variable} \wedge id \notin Var(Subst(op))\}$$

Var(*Subst*(*op*)) est l'ensemble des variables qui apparaissent dans le corps de l'opération. Les formules $id' == id$ représentent donc le prédicat avant-après pour les variables qui ne sont pas modifiées par l'opération *op*. La formule *F* est réécrite sous une forme normale disjonctive $\vee_i F_i$. Chaque F_i est un cas de test.

Différentes techniques d'extraction des cas de test peuvent être définies à travers l'attribut *xt*, qui est un prédicat. Si la valeur de *xt* est fixée à *true*, les cas de test sont obtenus par la mise sous forme normale disjonctive de la formule *F*. Nous obtenons de cette manière, les mêmes cas de test que ceux obtenus par la méthode de [Hörcher et Peleska 1995]. Cependant, l'extraction des cas de test peut être raffinée à l'aide de l'attribut *xt* en définissant par exemple, la partition de l'opérateur de disjonction logique proposée par [Dick et Faivre 1993], ou la décomposition des opérateurs de la théorie des ensembles. Par exemple, $e \in S_1 \cup S_2$ peut être décomposé en :

1) $e \in S_1 \cap S_2$ 2) $e \in S_1 - S_2$ 3) $e \in S_1 - S_2$.

Les cas de test F_i sont soumis à un outil de résolution de contraintes pour éliminer les cas qui contiennent des contradictions. Si, pour un cas de test, l'outil de résolution ne peut conclure ni à une contradiction ni à une solution, le cas de test est quand même conservé mais l'existence d'une solution n'est pas établie.

Pour la génération des séquences de test, un graphe est construit à partir de l'ensemble des cas de test. Les nœuds de ce graphe représentent des instanciations symboliques de l'état du système et les transitions sont l'application des cas de test. Contrairement à [Dick et Faivre 1993] et [Hierons 1997] on ne cherche pas à construire un automate qui modélise l'ensemble des séquences autorisées. On se contente d'une exploration partielle des valeurs d'états atteignables à partir de l'initialisation. Cette exploration s'arrête quand :

- tous les cas de test ont été appliqués au moins une fois ;
- aucun cas de test n'est applicable, ce qui peut être la conséquence d'une spécification incorrecte ;

- une limite fixée par l'utilisateur est atteinte (durée d'exécution autorisée, profondeur de l'exploration, nombre des cas de test déjà couverts, …).

Les chemins dans ce graphe représentent alors des séquences de test possibles. L'algorithme utilisé pour la sélection des séquences de test vise à choisir un ensemble minimal de chemins dans le graphe de manière à couvrir tous les cas de test.

2.4.2 Approches amont de formalisation du test

Nous venons de voir, dans le paragraphe précédent, des travaux appliqués à la construction de jeux de test fonctionnels à partir de spécifications orientées modèle. Il existe également dans la littérature des travaux plus amont, concernant la formalisation de certaines notions théoriques relatives au test dans le cadre des différentes notations. Nous mentionnons ici une approche de formalisation générique de stratégies de test dans la notation Z, et la définition d'une notion d'oracle pour le test d'un programme vis-à-vis de sa spécification B.

Les travaux présentés dans [Stocks et Carrington 1993; Carrington et Stocks 1994; Stocks et Carrington 1996] définissent un cadre formel pour spécifier des critères de test, et les cas qu'ils génèrent à partir des opérations d'une spécification Z. L'originalité de ces travaux est que les différentes notions relatives au test (critères, cas de test, oracle) sont exprimées dans la même notation (Z) que la spécification analysée. Le modèle formel d'un cas de test appelé *test template*. Un test template est un schéma Z qui décrit des contraintes sur les variables d'entrée d'une opération. Il définit donc un sous-ensemble du domaine d'entrée de l'opération. Un test template particulier est introduit comme le domaine d'entrée valide de l'opération : il correspond à la pré-condition de l'opération Z. Les critères de test sont alors modélisés en termes de stratégies d'analyse de partition : on exprime comment décomposer le domaine d'entrée de l'opération en sous-domaines. Les stratégies de décomposition appartiennent à type abstrait STRATEGY ; elles sont introduites par un nom et un ensemble de directives pour la génération des cas de test (par exemple, traitement de la disjonction logique, …). La décomposition effective en sous-domaines d'entrée est décrite par une fonction partielle qui, à partir d'un template et d'une stratégie, fournit un ensemble de templates. On obtient ainsi une arborescence de templates, dont la racine est le template décrivant le domaine d'entrée valide ; aux différents niveaux de l'arborescence, des templates fils sont dérivés d'un père par application d'une stratégie de décomposition. Des oracles partiels peuvent être associés à chaque test template. Les contraintes décrites sur les variables d'entrée sont utilisées pour extraire de l'opération des contraintes portant sur ses variables de sortie. Nous obtenons ainsi une description des sorties attendues de l'opération pour le test template considéré. Cette description constitue l'oracle.

Des exemples sont donnés dans [Stocks 1993], permettant d'illustrer l'expression de stratégies de test, et leur application à des exemples simples de spécifications Z. Notons que la notion de séquencement des cas de test n'est pas couverte par le cadre

formel proposé. Les auteurs signalent ce problème, mais argumentent que la définition formelle des cas de test devrait faciliter leur analyse et leur séquencement.

Des travaux ont été menés pour formaliser une procédure d'oracle dans le cas du test d'un programme par rapport à sa spécification B [Alnet 1996]. La spécification est un composant de type MACHINE. Comme nous l'avons présenté en détail dans le chapitre précédent, le comportement observable d'une machine abstraite est décrit dans le B-Book [Abrial 1996] à l'aide de substitutions externes. La notion de substitution externe est appelée une *trace* dans [Alnet 1996]. La sortie observable de la trace est l'ensemble des valeurs de sortie retournées par les opérations. La notion de *trace licite* prend en compte la terminaison des opérations. Une trace licite est une trace dans laquelle toutes les opérations sont appelées dans leur pré-condition. Dans le cas contraire, le comportement de la trace peut être quelconque à partir de l'opération dont la pré-condition n'est pas vérifiée : aucune procédure d'oracle n'est donc définie pour les traces illicites. La démarche proposée dans [Alnet 1996] est la suivante : soit un ensemble de traces licites construit pour le composant MACHINE ; pour chaque trace, le succès du test est établi si l'exécution du programme se termine et si l'ensemble des valeurs de sortie observées vérifie le prédicat avant-après de la trace.

2.5 Complémentarité du test et de la preuve

Dans les paragraphes précédents, nous avons présenté différentes approches de test structurelles et fonctionnelles. Le test est sans conteste la technique la plus utilisée pour la vérification du logiciel, et classiquement, les programmes testés sont issus d'un processus de développement informel. Dans le cadre d'un processus de développement formel, d'autres techniques de vérification, essentiellement la preuve, sont prescrites. On peut donc s'interroger sur la nécessité de maintenir des tests pour ce type de développement. Nous discutons, dans ce paragraphe, de la portée des méthodes formelles et du rôle laissé au test, d'abord de façon générale puis dans le cadre de la méthode B.

Pour la suite des discussions, il est nécessaire de pouvoir distinguer ce qui peut être formel et ce qui ne peut pas l'être. Nous reprenons les définitions suivantes qui sont extraites de [Gaudel 1991] : une *notation*, ou une *technique*, est formelle si elle est sujette à des manipulations systématiques suivant un calcul mathématique ; une *méthode* est formelle si elle utilise une notation ou une technique formelle.

2.5.1 Discussion générale

Plusieurs travaux dans la littérature discutent des limites des méthodes formelles dans le processus de développement du logiciel [Hall 1990; Bowen et Stavridou 1993; Bowen et Hinchey 1995; Gaudel 1995a].

La Figure 2.5, extraite de [Gaudel 1995a], fournit une vue simplifiée du processus de développement de logiciels, des différentes activités effectuées dans ce processus et des documents nécessaires à ces activités. Dans cette figure, les formes tracées représentent les documents et les flèches représentent les activités.

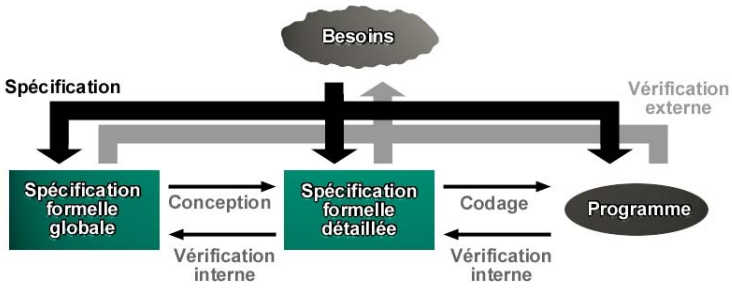


Figure 2.5. Schéma simplifié du processus de développement de logiciels

Les différents documents et activités montrés dans la Figure 2.5 peuvent être classés selon leur capacité à être formels ou non. Les besoins de l'utilisateur sont par essence informels, car il reflètent des aspects du monde réel. Les documents de la spécification peuvent être formels. Un programme peut, lui aussi, être considéré comme un document formel à condition qu'il existe une sémantique mathématique, c'est-à-dire une définition formelle du langage de programmation. Une activité est susceptible d'être entièrement formelle si les documents traités sont formels. Par conséquent, l'activité de spécification ne peut pas être formelle. Cependant, les activités de conception et de codage peuvent être formelles à condition d'avoir une spécification et un programme formels.

Une spécification formelle du service à délivrer peut être incorrecte pour deux raisons : les besoins de l'utilisateur peuvent être mal interprétés ou ces besoins peuvent être mal exprimés. La vérification de la conformité de la spécification aux besoins de l'utilisateur est appelée *vérification externe*[2]. Les besoins de l'utilisateur étant informels, cette activité ne peut pas être effectuée par la preuve. D'autres techniques de vérification, comme le test, doivent être utilisées. De plus, la spécification d'un système logiciel n'inclut pas toutes les caractéristiques de l'environnement réel. La spécification pose des hypothèses sur les composants matériels et logiciels qui interagissent avec le système. Ces hypothèses peuvent ne pas correspondre à la réalité et doivent être vérifiées. Le test est la seule technique permettant de vérifier le logiciel dans son environnement réel.

---

[2] Dans la littérature, cette activité est souvent désignée par le terme de *validation*. Conformément à la terminologie de [Laprie et al. 1996], qui donne à la validation un sens plus général, nous préférons adopter le terme de *vérification externe*, proposé par [Gaudel 1991].

L'activité de vérification externe ne concerne pas uniquement la vérification de la spécification globale. Les besoins de l'utilisateur peuvent contenir des besoins non-fonctionnels, par exemple des contraintes concernant une implémentation donnée, qui ne peuvent être pris en compte que dans les phases avancées du développement. La vérification externe doit donc accompagner tout le processus de développement, de la spécification au programme.

Partant d'une spécification donnée, l'activité de conception et de codage consiste à développer un programme : la *vérification interne*[3] a pour but de vérifier celui-ci par rapport à la spécification. Cette activité peut être entièrement formelle si le langage de spécification et le langage de programmation sont formels. On ne peut cependant vérifier que les propriétés exprimables dans le langage de spécification.

Mentionnons enfin le problème de la *validation de la validation* [Laprie et al. 1996], c'est-à-dire comment avoir confiance dans des méthodes et des outils utilisés pour avoir confiance dans le système. Par exemple, les preuves effectuées peuvent être incorrectes, ou l'expression des obligations de preuve à démontrer peut être incorrecte ; un générateur de code peut générer du code incorrect ; etc. Ce problème concerne donc tout le processus de développement.

2.5.2 Discussion dans le cadre de la méthode B

Le processus de développement en B est un processus formel, accompagné par la génération et la démonstration d'obligations de preuve. La question de l'allègement de certaines activités de vérification classiques, et notamment la suppression d'étapes de test, s'est donc posée à la communauté B. En opposition à cette tendance, [Waeselynck et Boulanger 1995] ont voulu réaffirmer l'importance du test. Dans leur argumentation, les auteurs mettaient d'abord en question la maturité des méthodologies de développement et des outils B (notamment le prouveur et les traducteurs en langage de programmation). Ensuite, ils soulevaient le problème de la vérification externe des modèles B.

Ces arguments demandent à être reconsidérés dans le contexte actuel. Depuis la publication de [Waeselynck et Boulanger 1995], les méthodologies de développement et les outils B se sont beaucoup améliorés. Des expériences d'utilisation de la méthode dans un contexte industriel (voir le chapitre précédent) ont amené les ingénieurs à définir des méthodologies de développement qui résultent en des architectures qui sont plus facilement prouvées. L'outil de preuve de l'Atelier B permet actuellement de démontrer automatiquement un pourcentage plus important d'obligations de preuve. Des démarches pour la validation des règles d'inférence utilisées par cet outil ont été également entreprises. Les traducteurs utilisés pour les

---

[3] Le terme de *vérification interne*, également proposé par [Gaudel 1991], est utilisé par opposition à la vérification externe. Dans la littérature, cette activité est souvent appelée *vérification*.

applications du domaine ferroviaire ont été couplés à la technique du monoprocesseur codé [Behm et al. 1997].

Il reste cependant quelques points faibles dans la méthode B. On peut citer le problème des obligations de preuve contenant des termes mal définis, c'est-à-dire des termes impliquant des fonctions partielles appelées en dehors de leur domaine de définition. Une solution théorique à ce problème est proposée dans [Behm et al. 1998; Burdy 2000], mais n'a pas encore été implémentée dans les outils B commercialisés. Un autre problème est celui des clauses de composition proposées par la méthode. La sémantique de ces clauses n'est pas définie formellement dans le B-Book. Une formalisation a été entreprise par [Bert et al. 1996; Potet et Rouzaud 1998; Rouzaud 1999]. Dans [Potet et Rouzaud 1998; Rouzaud 1999], les auteurs ont exhibé des exemples d'architectures telles que chaque composant B est prouvé, mais l'implémentation finale viole la spécification de départ. Les auteurs ont alors identifié des conditions architecturales restrictives permettant d'éviter ce problème : ces conditions ne sont pas toutes exigées par les outils B.

La vérification externe des modèles B reste un problème d'actualité. Dans les applications industrielles de la méthode B, les tests d'intégration avec le matériel et d'autres modules qui n'ont pas été développés en B sont maintenus. Ceci répond au problème de la vérification dans l'environnement réel. Par contre, la vérification de la spécification par rapport aux besoins fonctionnels s'effectue manuellement, par relecture des sources des composants B et analyse de traçabilité par rapport au cahier des charges. Cette vérification de la prise en compte des exigences fonctionnelles constitue clairement un point faible du processus. Nous allons donc nous y intéresser plus particulièrement. Ce problème nous semble d'autant plus important que les tests unitaires et d'intégration des modules développés en B sont supprimés.

Pour illustrer le problème de vérification externe des modèles B, nous utilisons dans le paragraphe suivant l'exemple d'un développement simple en B. Cet exemple nous permet de mettre en évidence la possibilité d'introduire des fautes dans la spécification qui ne sont pas révélées par les obligations de preuve.

2.5.3 Exemple introductif

L'exemple que nous allons utiliser est l'exemple du Triangle. Cet exemple est bien connu dans la communauté du test, et l'on peut trouver une spécification de cet exemple en langage Z dans [Carrington et Stocks 1994] et en VDM dans [Dick et Faivre 1992].

Le programme à spécifier reçoit en entrée trois valeurs entières censées représenter les longueurs des cotés d'un triangle. Il doit vérifier si ces valeurs peuvent définir un triangle. Si la réponse est positive, le programme doit calculer le type du triangle : scalène, isocèle, équilatéral ou rectangle. Des valeurs entières x, y et z peuvent définir un triangle si et seulement si elles vérifient les inégalités suivantes :

$$x < y + z \quad \wedge \quad y < z + x \quad \wedge \quad z < x + y$$

Le développement B de l'exemple du Triangle, que nous décrivons par la suite, a été écrit par un étudiant. Cet exemple est purement illustratif et a été choisi pour sa simplicité. Il permet de fournir un exemple des possibilités d'introduction de fautes à différentes étapes du développement, lorsque les besoins fonctionnels sont graduellement entrés dans la modélisation B. Il donne également une idée des mécanismes de raffinement et de décomposition en couches qui forment l'ossature d'un développement B. Enfin, il nous amène à poser la question de l'identification du sous-ensemble de composants censé constituer la spécification du problème.

La Figure 2.6 montre l'architecture du développement B du Triangle. Le composant de plus haut niveau est la MACHINE MAIN_INTERFACE. Il spécifie une opération main qui à ce niveau *ne fait rien* et est équivalente à skip. L'implémentation de cette opération dans MAIN_INTERFACE_1 consiste à demander à l'utilisateur d'entrer trois valeurs entières, à calculer la validité et le type du triangle, et à afficher le résultat. Le calcul de la validité et du type du triangle se fait par appel à l'opération Classe_Triangle, importée du composant TYPE_TRIANGLE. Le composant BASIC_TYPE_DE_TRIANGLE contient la définition de l'ensemble TYPE_DE_TRIANGLE = {ISOCELE, EQUILATERAL, SCALENE, RECTANGLE, INVALIDE} contenant les types de triangle, et les opérations de lecture et d'écriture de ces types.

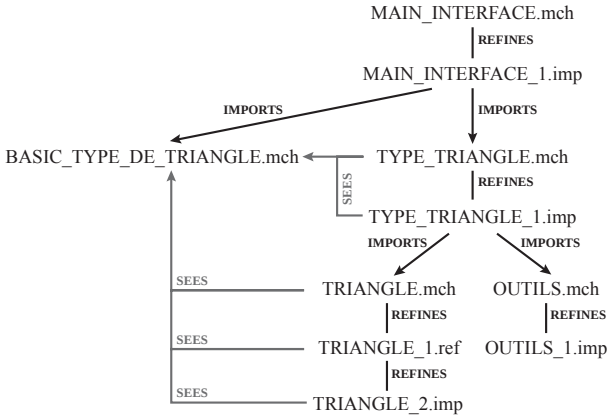


Figure 2.6. Architecture du développement de l'exemple du Triangle

La spécification de l'opération Classe_Triangle dans le composant TYPE_TRIANGLE est donnée dans la Figure 2.7.a. Cette spécification ne contient que du typage des paramètres d'entrée et de sortie de l'opération. Le raffinement de cette opération, proposé dans le composant TYPE_TRIANGLE_1 (voir la Figure 2.7.b), consiste à ordonner les valeurs d'entrées (opération Ordonne_3_NAT), à vérifier si ces valeurs constituent un triangle (opération Est_Un_Triangle) et, dans ce cas, à calculer le type de triangle (opération Quel_Triangle). Pour déterminer ce qui est réellement spécifié à ce niveau, examinons chacune de ces trois opérations.

```
lt ← Classe_Triangle (xx, yy, zz) ǐ
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧ zz ∈ NAT
      xx ≤ Max_Nat ∧  yy ≤ Max_Nat ∧
      zz ≤ Max_Nat
    THEN
      lt :∈ TYPE_DE_TRIANGLE
    END
```

(a) Spécification donnée dans le composant TYPE_TRIANGLE.mch

```
lt ← Classe_Triangle (xx, yy, zz) ǐ
    BEGIN
      VAR tx, ty, tz, un_bb IN
        tx,ty,tz ← Ordonne_3_NAT(xx,yy,zz);
        un_bb ← Est_Un_Triangle(tx,ty,tz);
        IF un_bb = TRUE
        THEN
          lt ← Quel_Triangle (tx, ty, tz)
        ELSE
          lt ← TYPE_COPY(INVALIDE)
        END
      END
    END
```

(b) Spécification donnée dans le composant TYPE_TRIANGLE_1.imp

```
uu, vv, ww ← Ordonne_3_NAT (xx, yy, zz) ǐ
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧  zz ∈ NAT
    THEN
      ANY ua, va, wa WHERE
        ua ∈ NAT ∧ va ∈ NAT ∧ wa ∈ NAT ∧ ua ∈ {xx, yy, zz} ∧
        va ∈ {xx, yy, zz} ∧ wa ∈ {xx, yy, zz} ∧ ua ≤ va ∧ va ≤ wa
      THEN
        uu, vv, ww := ua, va, wa
      END
    END
```

(c) Spécification donnée dans le composant OUTILS.mch

```
bb ← Est_Un_Triangle (xx, yy, zz) ǐ
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧
      zz ∈ NAT ∧ xx ≤ Max_Nat ∧
      yy ≤ Max_Nat ∧  zz ≤ Max_Nat ∧
      xx ≤ yy ∧ yy ≤ zz
    THEN
      ANY b_t
      WHERE b_t ∈ BOOL ∧
        (b_t = TRUE ⇒ yy > zz - xx)
      THEN bb := b_t
      END
    END;
```

```
letype ← Quel_Triangle (xx, yy, zz) ǐ
    PRE
      xx ∈ NAT ∧ yy ∈ NAT ∧
      zz ∈ NAT ∧ xx ≤ Max_Nat ∧
      yy ≤ Max_Nat ∧  zz ≤ Max_Nat ∧
      xx ≤ yy ∧ yy ≤ zz
    THEN
      ANY type
      WHERE type ∈ TYPE_DE_TRIANGLE
      THEN letype := type
      END
    END
```

(d) Spécifications données dans le composant TRIANGLE.mch

Figure 2.7. Spécification des opérations d'un sous-ensemble du développement du Triangle

L'opération Ordonne_3_NAT est spécifiée dans le composant OUTILS (voir la Figure 2.7.c). Cette spécification n'est pas assez précise vis-à-vis de la fonctionnalité qu'elle doit décrire. Par exemple pour les valeurs d'entrée *xx = 2*, *yy = 2* et *zz = 1*, quatre résultats possibles peuvent être acceptés par la spécification : (1) *uu = 1*, *vv = 1*, *ww = 1* ; (2) *uu = 1*, *vv = 1*, *ww = 2* ; (3) *uu = 1*, *vv = 2*, *ww = 2* et (4) *uu = 2*, *vv = 2*, *ww = 2*. Le résultat désiré est le numéro 3, mais les obligations de preuve ne garantissent pas que ce résultat sera effectivement choisi par les raffinements de cette opération.

Les deux opérations Est_Un_Triangle et Quel_Triangle sont spécifiées dans le composant TRIANGLE (voir la Figure 2.7.d). L'opération Est_Un_Triangle vérifie les inégalités de triangle. Étant donnée que les valeurs d'entrée sont ordonnées (ceci est spécifié dans la pré-condition de l'opération), la seule inégalité à vérifier est *yy* > *zz* - *xx*. Toutefois, la spécification de l'opération n'est pas correcte : le concepteur a utilisé l'opérateur logique d'implication au lieu d'équivalence. En conséquence, la spécification *bb* := FALSE peut être un raffinement correct de cette opération.

L'opération Quel_Triangle calcule le type d'un triangle valide. Cependant, la validité du triangle n'est pas exigée dans la pré-condition de l'opération. Le corps de cette opération ne contient que du typage des paramètres.

Ayant expliqué les opérations importées, nous pouvons maintenant conclure sur le raffinement de Classe_Triangle, proposé dans le composant TYPE_TRIANGLE_1. A cette étape du développement, les obligations de preuve garantissent que :

- toutes les opérations importées sont appelées dans leur pré-condition,
- après avoir expansé les appels aux opérations importées, le raffinement de Classe_Triangle préserve la sémantique de sa version plus abstraite, c'est-à-dire qu'étant appelé avec trois valeurs entières, il retourne une valeur appartenant à l'ensemble TYPE_DE_TRIANGLE.

Les besoins de l'utilisateur étant incomplètement exprimés par le modèle formel, il n'est pas garanti que les raffinements prouvés de Classe_Triangle délivreront le service attendu.

L'opération Classe_Triangle ne peut maintenant être raffinée qu'à travers les raffinements des opérations qu'elle importe. L'opération Quel_Triangle est la seule opération dont le raffinement est réalisé en deux étapes. Le raffinement de cette opération dans le composant TRIANGLE_1 précise le résultat pour les triangles de type équilatéral et isocèle, mais reste imprécis pour les types rectangles et scalène.

Dans cet exemple, la spécification du problème n'est donc obtenue qu'à la fin du développement. Des fautes peuvent être introduites à n'importe quelle étape du développement formel, sans qu'elles soient révélées par les obligations de preuve.

Cet exemple constitue un cas extrême. Dans des exemples donnés dans [Lano et Haughton 1996], les phases de spécification et de raffinement sont clairement séparées : la spécification ne contient pas de composants REFINEMENT ni IMPLEMENTATION, uniquement des composants MACHINE. Toutefois, pour une application réelle, une telle séparation n'est pas faisable. Afin de faciliter les preuves,

plusieurs étapes de raffinement et de décomposition sont nécessaires pour rentrer les besoins graduellement dans le modèle. Les approches de modélisation utilisées dans l'industrie ferroviaire [Behm et al. 1997] procèdent en deux phases. On spécifie d'abord tous les besoins fonctionnels dans une phase de conception préliminaire. Le modèle obtenu à l'issue de cette phase constitue un sous-ensemble de l'architecture finale, et comporte plusieurs étapes de raffinement et de décomposition en couches. Ce modèle est ensuite raffiné pour obtenir des versions de plus en plus concrètes, jusqu'à la génération du code exécutable. Sous réserve que le modèle intermédiaire exprime les besoins attendus, nous pouvons faire confiance aux obligations de preuve pour garantir la correction des raffinements ultérieurs.

Nous proposons d'utiliser des méthodes de test pour vérifier la conformité de la spécification par rapport aux besoins fonctionnels. Normalement, le test concerne la dernière étape de développement, c'est-à-dire l'étape à laquelle un programme est généré. Cependant, comme il a été souligné par [Bowen et Hinchey 1995; Gaudel 1995a], des tests peuvent être effectués pendant les étapes intermédiaires du développement, à condition d'avoir une spécification *exécutable*. Ce que l'on entend par *exécution* est de pouvoir déduire automatiquement de la spécification les sorties associées à des valeurs d'entrée. Par la suite, conformément à la littérature, nous utiliserons indifféremment les termes d'exécution ou d'*animation* des spécifications.

2.6 Animation des spécifications

Les spécifications exécutables permettent d'utiliser les techniques de vérification dynamique, en plus des techniques statiques, pour trouver des fautes de spécification. Dans la littérature, deux points de vue s'opposent quant au caractère exécutable ou non des spécifications formelles. Nous discutons de ces points de vue dans le paragraphe 2.6.1. Le paragraphe 2.6.2 est consacré aux travaux qui portent sur l'animation des spécifications orientées modèle. Des travaux existent également pour formaliser la notion de correction de l'animation. Nous les exposons dans le paragraphe 2.6.3.

2.6.1 Spécifications exécutables et non-exécutables

Dans [Hayes et Jones 1989], quelques arguments contre le principe de spécification exécutable sont exposés. Ils considèrent quelques classes de problème et montrent à l'aide d'exemples que des techniques de spécification, comme l'indéterminisme et les quantificateurs, sont à la fois utiles et difficiles à rendre exécutable. Leurs arguments peuvent être résumés par les deux points suivants :

- le choix d'un formalisme pour la tâche de spécification doit être guidé par son pouvoir d'expression et non par le fait qu'il est exécutable ;

- une spécification exécutable contient des détails algorithmiques qui ne sont pas souhaitables. Ces détails rendent la tâche de spécification plus difficile. Ils conduisent à sur-spécifier le problème et par ce fait à écarter des choix d'implémentation éventuellement plus efficaces. De plus, la vérification de l'implémentation par rapport à une spécification qui contient des détails algorithmiques est plus difficile.

Ces arguments sont réfutés dans [Fuchs 1992] pour deux raisons. Tout d'abord l'auteur considère que le problème le plus important dans le développement du logiciel est sa correction et pas l'éventuel manque d'expressivité des langages de spécification. Les spécifications exécutables permettent, en plus d'effectuer des raisonnements formels, d'utiliser des méthodes de vérification dynamique et d'effectuer un prototypage rapide qui permet aux utilisateurs et aux développeurs d'avoir une vue globale du comportement du système. En second lieu, l'auteur considère que l'expressivité et l'exécutabilité ne s'excluent pas forcément, si les spécifications sont écrites dans un langage déclaratif. En effet, l'auteur traduit tous les exemples de [Hayes et Jones 1989], sans changement essentiel de leur structure, dans un langage de spécification logique semblable à Prolog.

Comme [Fuchs 1992], nous adhérons à l'idée que l'animation fournit un moyen utile pour la vérification des spécifications. La nature non-exécutable des théories mathématiques à l'origine des langages de spécification, ne contredit pas le fait que des spécifications puissent être animées. Dans [Gravell et Henderson 1996], les auteurs mentionnent le grand nombre de spécifications déjà publiées qui ont été animées par la suite avec succès. Ils en déduisent qu'il est possible d'animer, au moins dans une certaine mesure, des spécifications réelles. Nous notons également le nombre important d'articles concernant les techniques et les outils d'animations des spécifications (voir le paragraphe suivant).

2.6.2 Techniques et outils d'animation

Dans ce paragraphe, nous allons passer en revue les principaux travaux et outils concernant l'animation des spécifications orientées modèle Z, VDM et B.

De nombreux travaux existent concernant l'animation et le prototypage des spécifications Z dans différents langages, notamment Prolog [Dick et al. 1989; West et Eaglestone 1992]. Les constructions du langage Z, c'est-à-dire les opérateurs de la théorie des ensembles, sont implémentées dans une bibliothèque de codes Prolog. En utilisant les clauses de cette bibliothèque, les schémas Z sont ensuite traduits en programme Prolog. On hérite de cette manière de la stratégie d'exécution de Prolog : les clauses sont évaluées dans l'ordre de leur écriture. Cette stratégie n'étant pas très performante, les auteurs ont souvent trouvé nécessaire de modifier les clauses Prolog obtenues à partir de la traduction des schémas Z, pour améliorer le temps d'exécution. Ces modifications consistent le plus souvent en un re-ordonnancement des clauses Prolog. Elles ne sont pas automatiques, mais dans [Dick et al. 1989] les auteurs

mentionnent l'utilisation d'un outil qui permet d'appliquer des règles de transformation choisies manuellement.

Des travaux plus récents concernent l'animation des spécifications écrites en Sum, une extension du langage Z [Hazel et al. 1997; Hazel et al. 1998]. Les spécifications ici ne sont pas traduites, elles sont directement interprétées par l'outil d'animation. L'outil, appelé Possum, est développé en Qu-Prolog, une variante de Prolog. Lors de l'évaluation du prédicat avant-après d'une opération, l'outil réalise un ordonnancement de clauses à l'aide d'heuristiques.

Pour l'animation des spécifications B, on peut distinguer deux types d'approche. La première est celle des outils commerciaux *Atelier B* et *B Toolkit*, qui proposent chacun leur propre outil d'animation. Ces outils sont basés sur des principes similaires. Ils fournissent des commandes permettant d'exécuter des opérations d'un composant B de type MACHINE. L'outil d'animation utilise des règles de réécriture pour simplifier les prédicats et les expressions en valuant les variables. En cours d'exécution, chaque fois qu'une substitution indéterministe est rencontrée, l'animation est interrompue. L'utilisateur est alors chargé de lever l'indéterminisme. Dans le cas de l'*Atelier B*, l'outil d'animation étant lié à l'outil de preuve, il permet de faire des preuves en ligne, par exemple pour vérifier la satisfaction d'une pré-condition.

La deuxième approche est celle rapportée dans [Py et al. 2000] concernant l'animation des spécifications B en utilisant les techniques de programmation logique avec contraintes ensemblistes. Cette approche correspond à une exécution symbolique de la spécification. Ainsi, les variables d'état de la spécification sont représentées par un système de contraintes et non pas par un ensemble de valeurs. Dans ce contexte, l'exécution des opérations est simulée par la réduction des contraintes. Avant d'animer les opérations, elles sont traduites sous forme de clauses du langage de programmation logique. Cette technique d'animation permet de conserver l'indéterminisme et de diminuer le nombre d'états générés, puisqu'un état représenté par un ensemble de contraintes peut correspondre à plusieurs états valués. Comme dans le cas des outils B du commerce, l'animation est limitée à un composant de type MACHINE et ne traverse pas les couches de raffinement.

Pour conclure sur les outils liés à l'animation des spécifications, mentionnons également deux approches moins puissantes que les précédentes. Dans [Mikk 1995], pour évaluer une opération écrite en Z l'utilisateur doit donner une valeur aux paramètres d'entrée, aux paramètres de sorite, et aux variables d'état. La conformité de ces valeurs avec la spécification est ensuite vérifiée par l'outil. Les travaux de [Kans et Hayton 1994] réalise un prototypage rapide d'un sous-ensemble de VDM en utilisant le langage impératif ABC, une extension du langage BASIC. Cette approche est proposée dans un but pédagogique et ne vise pas à traiter des spécifications réalistes.

2.6.3 Problème de la correction de l'animation

Breuer et Bowen se sont intéressés au problème de la correction d'une animation de Z [Breuer et Bowen 1994]. Ils donnent d'abord une définition de la correction, puis ils utilisent cette définition pour prouver la correction d'un exemple d'interpréteur Z. Informellement, la correction doit garantir que si un résultat est obtenu lors d'une animation, il doit être une approximation du résultat attendu exprimé dans la logique du premier ordre et la théorie des ensembles. Il est à noter que la définition de la correction ne signifie pas nécessairement que l'animation doive toujours se terminer. Si l'animation couvre une portion significative de la grammaire Z, la non-terminaison doit être un comportement prévu.

Pour formaliser la non-terminaison, Breuer et Bowen introduisent la notation de sortie indéfinie, $\perp$. Le résultat correct d'une animation peut alors être : 1) indéfini ($\perp$) – c'est-à-dire l'outil ne donne aucun résultat et le calcul ne se termine pas, 2) partiel – c'est-à-dire l'outil donne un sous-ensemble de résultats possibles, mais le calcul ne se termine pas, 3) une approximation exacte – c'est-à-dire le calcul termine en donnant l'ensemble des résultats attendus.

Cette définition de la correction couvre les notions de spécification imprécise et de spécification miraculeuse. Si la spécification est imprécise (indéterminisme), l'approximation exacte est l'ensemble des résultats possibles, et si la spécification est miraculeuse (impossible à satisfaire), l'approximation exacte est l'ensemble vide. Ces notions sont formalisées dans [Breuer et Bowen 1994].

2.7 Conclusion

Le processus de développement en B est un processus formel. Il procède par raffinements successifs d'un modèle abstrait vers des modèles de plus en plus concrets, jusqu'à la génération de code dans un langage de programmation. Pour chaque étape du raffinement, la méthode B génère des obligations de preuve qui garantissent la correction de ce raffinement. En supposant la correction des preuves effectuées, ces obligations de preuve constituent la vérification interne d'un développement en B. En revanche, la vérification externe, qui n'est pas dans la portée d'une méthode formelle, nous intéresse plus particulièrement dans le cadre de nos travaux.

Nous proposons d'utiliser les techniques de test pour révéler les fautes de spécification qui sont dues soit à une mauvaise compréhension d'un besoin fonctionnel, soit à une modélisation incorrecte de celui-ci.

Si la spécification est exécutable, on peut procéder au test d'une manière classique, c'est-à-dire générer les entrées de test et vérifier les sorties. Dans le paragraphe 2.6.2, nous avons mentionné l'existence d'outils d'animation dans les ateliers commerciaux de la méthode B, et nous avons également discuté de travaux académiques visant à améliorer les techniques d'animation de ces spécifications.

Nous définirons, dans le troisième chapitre de ce mémoire, un cadre formel pour le test de modèles B. Ce cadre prend en compte les spécificités du développement formel B (mécanismes de raffinement et de composition de machines abstraites) et formalise certaines notions relatives au test. Comme [Stocks et Carrington 1996], nous adopterons le principe d'une formalisation où les notions introduites s'expriment dans la même notation que celle de la spécification analysée.

Le chapitre 4 sera consacré à la définition de critères de couverture du texte d'une spécification B. La notation B couvrant un large spectre, des abstractions aux implémentations, nous nous inspirerons à la fois des critères définis pour les spécifications Z et VDM, et des critères structurels définis pour les programmes.

Cadre théorique pour le test de modèles B

3.1 Introduction

Dans le chapitre précédent, nous avons discuté du rôle du test dans le processus de développement formel B. Nous avons mis l'accent sur le problème de la vérification externe et avons souligné que cette vérification ne concerne pas uniquement l'intégration du logiciel avec le matériel et les modules externes, mais également la vérification de la spécification par rapport aux exigences fonctionnelles décrites dans le cahier des charges.

Nous présentons, dans ce chapitre, le cadre théorique que nous avons défini pour le test des modèles B [Behnia et Waeselynck 1998; Waeselynck et Behnia 1998]. Nous commençons, dans le paragraphe 3.2, par les motivations qui nous ont amenés à définir ce cadre. Le cadre théorique prend en compte les spécificités du développement formel en B, c'est-à-dire les mécanismes de raffinement et de composition de machines abstraites, pour répondre à deux questions essentielles :

- dans le processus incrémental de développement en B, quels sont les modèles qui peuvent être testés ?
- comment définir le test de ces modèles ?

Les réponses à ces questions sont exposées respectivement dans les paragraphes 3.3 et 3.4. Dans le paragraphe 3.5, nous discutons des conséquences et des limites du cadre de test ainsi défini. Nous présentons ensuite deux exemples permettant d'illustrer et de mettre en pratique les concepts liés à ce cadre : l'exemple du Triangle (paragraphe 3.6) qui avait été introduit dans le chapitre 2, puis un exemple industriel de développement B dans le domaine ferroviaire (paragraphe 3.7). Nous terminons, dans

le paragraphe 3.8, par quelques règles méthodologiques pouvant être considérées lors du développement d'applications en B, en vue de leur testabilité.

3.2 Motivations pour un cadre unifié de test

Notre objectif est d'utiliser des techniques de test pour révéler des fautes de spécification qui sont dues soit à une mauvaise compréhension d'un besoin fonctionnel, soit à une mauvaise modélisation d'un besoin bien compris. Cette vérification est d'autant plus efficace qu'elle est effectuée tôt dans le développement, avant la génération de code exécutable. L'animation de sources B semble une possibilité prometteuse pour valider les modèles abstraits, d'où l'idée de définir un cadre *unifié* de test indépendant du fait que les jeux de test soient exécutés sur les modèles abstraits ou sur le code final généré.

Nous pouvons considérer un processus de développement B comme une série d'étapes durant lesquelles des modèles de plus en plus concrets de l'application sont construits. La Figure 3.1 schématise cette vue simplifiée du développement. Les modèles obtenus à différentes étapes de développement sont liés par la relation de raffinement, et les obligations de preuve (OPs) générées par la méthode B garantissent la correction de chaque raffinement.

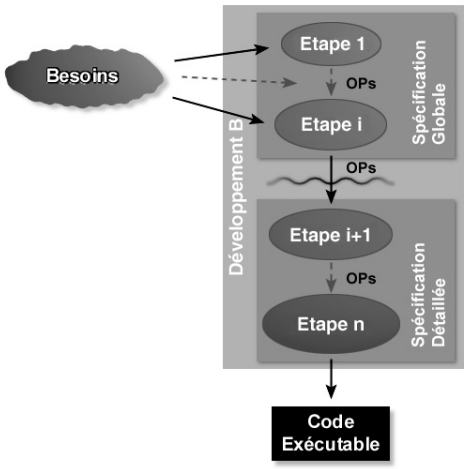


Figure 3.1. Schéma général de développement formel en B

Supposons qu'à une étape de développement, on obtienne un modèle censé spécifier tous les besoins fonctionnels de l'utilisateur (par exemple le modèle obtenu à l'étape *i* de la Figure 3.1). Avant de poursuivre le raffinement vers une implémentation, il est souhaitable de vérifier que ce modèle prend correctement en compte ces besoins. Sous réserve que les propriétés vérifiées correspondent à des

propriétés préservées par raffinement, il n'est alors pas nécessaire de procéder à la vérification externe des modèles obtenus aux étapes ultérieures.

L'étape à laquelle tous les besoins de l'utilisateur ont été rentrés peut se situer n'importe où dans le développement B. Comme nous l'avons souligné dans le Chapitre 1, le raffinement n'est pas seulement utilisé pour obtenir des versions de plus en plus concrètes de l'application. Il est également utilisé comme une technique de spécification, pour rentrer au fur et à mesure les besoins de l'utilisateur dans le modèle. Si certains besoins ne sont rentrés qu'à la fin du développement, leur vérification peut s'effectuer par test du code final, ce dernier constituant une version compilée du modèle le plus concret. D'un point de vue méthodologique, il est cependant préférable d'exprimer les besoins plus tôt dans le développement. Ceci correspond également à la pratique industrielle dans laquelle les phases de spécification globale et spécification détaillée sont séparées [Dehbonei et Mejia 1995; Behm et al. 1997; Behm et al. 1999]. On peut alors effectuer la vérification plus tôt dans le développement, à condition d'avoir un moyen de tester les modèles abstraits.

Plusieurs outils d'animation de spécifications orientées modèle existent déjà, dont quelques-uns pour la notation B (voir chapitre 2). Ceci nous a amenés à l'idée de proposer un cadre théorique *unifié* de test, indépendant du niveau d'abstraction du modèle sous test. Ce cadre théorique doit prendre en compte les obligations de preuve qui accompagnent le raffinement en B : pour un même jeu de test, l'acceptation des résultats fournis par un modèle doit impliquer l'acceptation des résultats fournis par des raffinements prouvés de ce modèle.

Le cadre théorique que nous avons défini comprend deux aspects :

- Identification des étapes de développement, auxquelles des modèles de l'application sont obtenus. Ce sont ces modèles qui pourront faire l'objet du test.
- Définition formelle des notions de séquence de test et d'oracle. Cette définition tient compte de la notion de correction introduite par le raffinement.

Nous allons discuter de chacun de ces aspects dans les paragraphes 3.3 et 3.4.

3.3 Identification des étapes de développement

3.3.1 Position du problème

Un développement B consiste en la construction incrémentale d'une architecture de composants liés par des liens de raffinement (REFINES), de décomposition (IMPORTS) ainsi que de composition et de partage (INCLUDES, SEES et USES). La Figure 3.2 présente un exemple d'une telle architecture, pour un projet B hypothétique. Dans cette figure, nous avons identifié des sous-ensembles de composants, que nous faisons correspondre à des étapes de développement ordonnées. Cependant, la notion d'étape de développement n'est pas définie dans la méthode B. En toute rigueur, rien ne nous

autorise à extraire, du graphe de l'architecture, cette représentation du processus de développement en termes d'une succession de modèles de plus en plus concrets. Nous devons donc préciser comment identifier les étapes de développement, et justifier la relation de raffinement entre modèles correspondants.

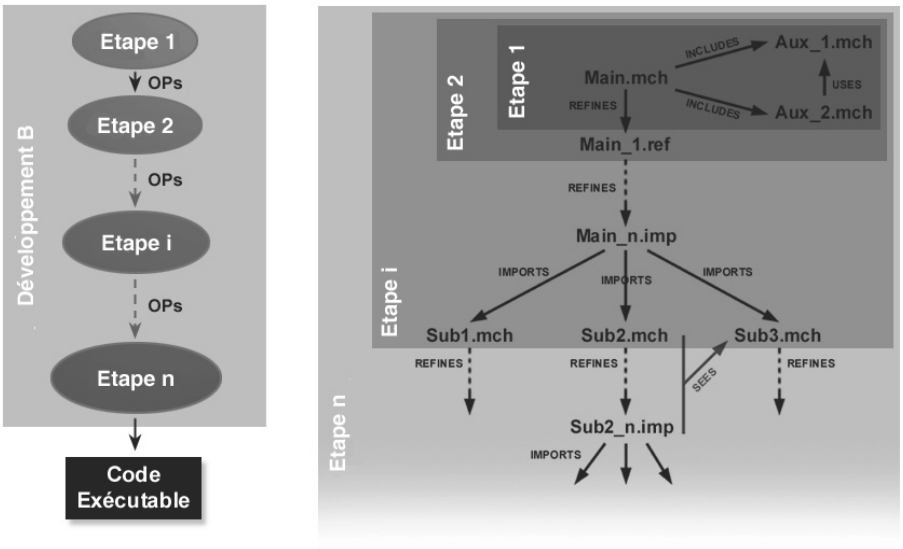


Figure 3.2. Identification des étapes de développement

L'identification des étapes de développement que nous proposons est basée sur la notion de dépliage. Le dépliage consiste à combiner syntaxiquement les textes formels d'un ensemble de composants, pour produire un texte formel unique, dans lequel les liens de construction (REFINES, IMPORTS, …) ont été éliminés par mise à plat. Nous avons mentionné un exemple de ce type de combinaison syntaxique pour le lien REFINES dans le paragraphe 1.4.3.

Pour qu'un ensemble de composants puisse constituer une étape de développement, deux types de conditions doivent être vérifiées :

1. Le texte formel aplati construit à partir de cet ensemble doit décrire une machine abstraite. En effet, en B, seules les machines abstraites offrent un comportement observable qui peut être testé, et seules les machines abstraites peuvent être comparées selon la relation de raffinement. Les conditions architecturales liées à la construction de machines abstraites aplaties dépendent des règles syntaxiques et de visibilité des liens de composition en B. Nous identifierons ces conditions dans le paragraphe 3.2.2.
2. La relation de raffinement doit être établie entre machines abstraites aplaties. En particulier, toute machine obtenue à une étape intermédiaire *i* doit être une abstraction de la machine correspondant à la dernière étape *n*, qui sera compilée en

code exécutable. Dans le cas contraire, il n'y aurait aucune garantie que le comportement modélisé par la machine *i* soit représentatif (selon la relation de raffinement) de celui de l'implémentation finale : on ne pourrait rien déduire du test de cette machine *i*. La relation de raffinement n'est pas une simple relation d'inclusion entre ensembles de composants prouvés : des conditions additionnelles doivent être vérifiées. Nous discuterons de ces conditions dans le paragraphe 3.2.3.

L'algorithme de dépliage proprement dit sera présenté au paragraphe 3.2.4.

Les deux types de conditions énoncés ci-dessus fondent la correspondance montrée dans la Figure 3.2. Pour cet exemple, le modèle (machine aplatie) le plus abstrait de l'application correspond au composant Main.mch enrichi avec les textes formels de Aux_1.mch et Aux_2.mch. Le modèle le plus concret correspond à la mise à plat de la totalité de l'architecture. Des modèles intermédiaires sont obtenus lors d'étapes introduisant des liens de raffinement (REFINES) et de décomposition en couches (IMPORTS). Nous verrons que cette analyse d'une architecture B peut également être appliquée à des sous-systèmes de l'application, par exemple le sous-système dont la première étape de développement est constituée du composant Sub2.mch. Dans les modèles de sous-systèmes, des liens SEES peuvent subsister après dépliage, car pointant vers des composants de l'application extérieurs au sous-système considéré. Ces modèles peuvent être testés, mais l'environnement de test devra fournir un bouchon (*stub*) pour chaque composant vu.

Notre cadre théorique permet donc l'identification systématique d'étapes de développement pour l'application complète, ou pour un sous-système de celle-ci. Par contre, la détermination de l'étape à partir de laquelle toutes les exigences fonctionnelles ont été rentrées, par exemple l'étape *i* dans la Figure 3.1, n'est pas automatisable. Ceci pose le problème du choix du modèle à tester. Tous les modèles obtenus par notre approche exhibent un comportement observable qui peut être testé. Mais les modèles correspondant aux premières étapes de développement sont généralement trop incomplets pour exhiber un comportement significatif vis-à-vis des besoins que l'on cherche à vérifier. Le choix d'un modèle peut être guidé par une analyse de traçabilité des exigences du cahier des charges. Si le modèle retenu est encore trop imprécis, ce sont alors les résultats du test qui pourront mettre en évidence la nécessité de rejouer certaines séquences de test à une étape ultérieure.

Après cette présentation générale du problème de l'identification des étapes de développement, nous allons maintenant détailler les solutions apportées par nos travaux.

3.3.2 Conditions pour la construction de machines abstraites aplaties

Une architecture de développement en B peut être représentée par un graphe. Les nœuds de ce graphe sont les composants du développement, et ses arcs représentent les différents liens entre les composants (REFINES, IMPORTS, SEES, INCLUDES et USES). Les composants explicitement renommés lors d'une importation ou d'une inclusion, sont représentés par des nœuds distincts dans le graphe. Un nœud particulier du

graphe est constitué du composant MACHINE qui spécifie l'opération *main* du projet. Soit M ce composant. Le graphe de l'architecture est le quadruplet $\mathcal{G} = <M, \mathcal{N}, \mathcal{E}, \lambda>$, où $\mathcal{N}$ est l'ensemble des noms des composants du projet, $\mathcal{E}$ est l'ensemble des arcs connectant des couples de composants, et $\lambda \in \mathcal{E} \rightarrow$ {est_raffiné_par, importe, voit, inclut, utilise} est une fonction totale donnant une étiquette à chaque arc. Les arcs de $\mathcal{G}$ sont étiquetés selon les liens spécifiés entre composants. Par exemple :

- Pour tout n_i, $n_j \in \mathcal{N}$, si le composant désigné par n_j référence le composant désigné par n_i dans sa clause REFINES, alors $(n_i, n_j) \in \mathcal{E}$ et $\lambda(n_i, n_j)$ = est_raffiné_par.
- Pour tout élément n_i et n_j de $\mathcal{N}$, si le composant désigné par n_i référence n_j dans sa clause IMPORTS, alors $(n_i, n_j) \in \mathcal{E}$ et $\lambda(n_i, n_j)$ = importe.

On procède de même pour les arcs étiquetés voit, inclut et utilise.

A partir du graphe complet, une structure d'arbre $\mathcal{T}$ peut être extraite. Cette structure d'arbre correspond à l'ossature de l'architecture et ne contient explicitement que les liens de raffinement et d'importation. Il s'agit bien d'un arbre, car l'architecture du projet doit être conforme aux exigences de la méthode B : les liens REFINES et IMPORTS ne contiennent pas de cycle, chaque composant est raffiné par seulement un composant et un composant ne peut être importé plus d'une fois avec le même nom. La racine de $\mathcal{T}$ est le composant M. Les nœuds de $\mathcal{T}$ forment un sous-ensemble des nœuds de $\mathcal{G}$: les nœuds manquants correspondent à des composants qui ne sont pas importés ni raffinés, et ne servent qu'à l'enrichissement local (via le lien INCLUDES) des textes formels des composants de $\mathcal{T}$. Par exemple, dans la Figure 3.2, les composants Aux_1 et Aux_2 ne servent qu'à l'enrichissement du texte du Main, et n'appartiennent pas à l'ossature de l'architecture.

En nous basant sur cette structure arborescente, nous allons définir les conditions architecturales qui doivent être vérifiées par un ensemble de composants afin que leur mise à plat puisse résulter en une machine abstraite. La fonction Φ de dépliage, dont nous détaillerons l'algorithme dans le paragraphe 3.3.4, prend en entrée un sous-arbre de $\mathcal{T}$ représentant un ensemble de composants B à déplier. Cet ensemble comprend : 1) les nœuds du sous-arbre, 2) les composants transitivement inclus par chacun de ces nœuds. En parcourant récursivement les nœuds du sous-arbre, Φ met à plat tous les liens entre composants de cet ensemble, y compris les liens SEES et USES entre composants importés ou inclus. Par combinaison syntaxique des textes formels des composants, on produit ainsi le texte formel déplié équivalent au sous-ensemble de l'architecture considéré. Tout sous-arbre de $\mathcal{T}$ ne correspond pas à un ensemble de composants dépliable. Nous allons d'abord définir le domaine d'entrée valide de Φ : les conditions architecturales correspondantes garantissent l'obtention d'un texte formel déplié. L'ajout d'une condition supplémentaire nous permettra alors de garantir que ce texte formel est celui d'une machine abstraite.

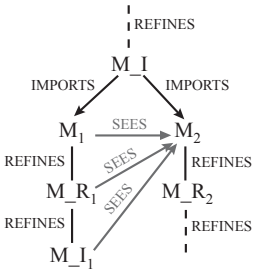
Le domaine d'entrée valide de Φ est l'ensemble $\mathcal{ST}$ des sous-arbres de $\mathcal{T}$ qui respectent trois conditions architecturales données ci-dessous.

Les deux premières conditions concernent le lien d'importation. Dans un composant IMPLEMENTATION, les opérations et les données des composants importés sont visibles. Pour construire le texte formel déplié du composant IMPLEMENTATION,

nous avons besoin du texte formel de tous les composants importés. Ainsi, si un composant IMPLEMENTATION fait partie du sous-arbre à déplier, soit il n'importe aucun composant, soit tous les composants importés font également partie du sous-arbre. Pour tout sous-arbre $T \in \mathcal{ST}$, ces conditions sont décrites comme suit :

1. une IMPLEMENTATION qui n'est pas une feuille dans $\mathcal{T}$ ne peut être une feuille dans T ;
2. si un nœud N n'est pas une feuille dans T, alors tous ses fils dans $\mathcal{T}$ font partie de T.

Notre troisième condition concerne le lien SEES. Les règles de visibilité sont différentes si un composant est vu par un composant de type MACHINE ou REFINEMENT, ou par un composant de type IMPLEMENTATION. Nous montrons dans la Figure 3.3 ces règles pour un sous-arbre de composants. Le composant *M_I* ne peut pas être implémenté sur M_1 et un raffinement de M_2 : à cause des règles de visibilité du lien SEES, M_1 peut consulter les variables abstraites de M_2 qui ne seront plus présentes dans M_R_2, rendant impossible la combinaison syntaxique de ces composants. Ainsi, si M_1 est une feuille du sous-arbre considéré, aucun raffinement de M_2 ne peut être présent dans ce sous-arbre.



	variables de M_2	opérations de M_2
opérations de M_1	visible	visible
opérations de M_R_1	visible	visible
opérations de M_I_1	non-visible	visible

Figure 3.3. Règles de visibilité du lien SEES pour les composants MACHINE, REFINEMENT, ou IMPLEMENTATION

Plus généralement, pour tout sous-arbre $T \in \mathcal{ST}$, cette condition peut être décrite de la manière suivante :

3. pour chaque feuille N_i de T correspondant à un composant MACHINE ou REFINEMENT, si N_i voit un composant N_j, ou N_i inclut (transitivement) un composant qui voit N_j, et si N_j n'est pas (transitivement) inclus dans N_i, alors soit N_j n'est pas dans T, soit N_j est une feuille de T.

Le texte formel déplié $\Phi(T)$ ne contiendra plus de liens INCLUDES, USES ou IMPORTS : d'après la définition de $\mathcal{ST}$, tous ces liens sont internes à l'ensemble de composants représenté par T, et ont été mis à plat. Il peut cependant rester une clause REFINES ou SEES. Si $\Phi(T)$ contient une clause REFINES, il n'est pas encore le texte d'une machine abstraite, mais un texte intermédiaire destiné à enrichir un composant de type MACHINE. Nous ajoutons alors une dernière condition architecturale :

4. le nœud racine de T doit désigner un composant de type MACHINE.

Par construction, $\Phi(T)$ ne contient pas de clauses SEES si tout composant vu depuis un nœud de T est également importé dans T : $\Phi(T)$ constitue alors le texte d'une machine abstraite indépendante. Mais ceci n'est imposé par aucune des conditions ci-dessus. On se laisse la possibilité de construire des modèles d'un sous-système de l'application, pouvant éventuellement contenir des liens SEES vers des composant extérieurs à ce sous-système. Nous allons cependant voir que la présence de liens SEES dans des modèles aplatis ne doit être autorisée que dans certains cas restrictifs, si l'on veut pouvoir ordonner ces modèles selon la relation de raffinement.

3.3.3 Ordonnancement des étapes de développement

Dans le paragraphe précédent, nous avons défini quatre conditions concernant la construction de machines abstraites. Ainsi, pour tout sous-arbre $T \in \mathcal{ST}$ dont la racine est un composant de type MACHINE, $\Phi(T)$ constitue une machine abstraite. Pour que les modèles ainsi construits puissent correspondre à une étape de développement, nous devons pouvoir les ordonner selon la relation de raffinement.

Nous définissons d'abord une relation d'ordre partiel sur $\mathcal{ST}$. Soient T_1 et T_2 deux éléments de $\mathcal{ST}$, $T_1 \leq T_2$ si et seulement si T_1 et T_2 ont le même nœud racine, qui est un composant de type MACHINE, et T_1 est un sous-arbre de T_2. La question se pose alors de savoir si l'ordonnancement des sous-arbres implique l'ordonnancement des modèles correspondants : a-t-on $T_1 \leq T_2 \Rightarrow \Phi(T_1) \lozenge \Phi(T_2)$? Malheureusement, la réponse est négative, comme nous le verrons au paragraphe 3.3.3.1. Nous devons donc identifier une condition suffisante pour garantir $\Phi(T_1) \lozenge \Phi(T_2)$. Pour cela, nous distinguerons deux cas : 1) les modèles construits à partir de la racine de l'architecture (dans le paragraphe 3.3.3.2) et 2) les modèles construits à partir de la racine d'un sous-système, qui peuvent donc avoir des liens SEES vers d'autres sous-systèmes (dans le paragraphe 3.3.3.3).

3.3.3.1 Effets de bord dus à la clause SEES

Le problème vient des liens de partage introduits par la clause SEES, qui peuvent induire des phénomènes d'aliasing, d'où d'éventuels effets de bord. Pour illustrer ce problème, reprenons l'exemple proposé par [Rouzaud 1999]. La Figure 3.4 donne l'architecture de cet exemple. Nous allons l'analyser en termes d'ordonnancement d'arbres et de modèles, en commentant brièvement les opérations spécifiées dans les différents composants.

Soit T_1 le sous-arbre constitué du seul composant A_1. Ce composant spécifie une opération Op_1 qui retourne toujours la valeur TRUE.

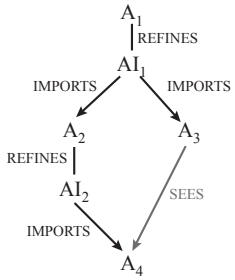


Figure 3.4. Architecture de l'exemple extrait du [Rouzaud 1999]

Considérons maintenant le sous-arbre T_2 de racine A_1 et de feuilles A_2 et A_3. L'implémentation de l'opération Op_1 dans AI_1 est la suivante :

$rr \leftarrow Op_1$ ĩ VAR *V1*, *V2* IN
V1 ← Val_3; Op_2; *V2* ← Val_3;
IF *V1* = *V2* THEN *rr* := TRUE ELSE *rr* := FALSE END
END

Le dépliage consiste notamment à expanser les appels d'opérations. L'opération Op_2 est importée du composant A_2, et ne fait rien : Op_2 ĩ skip. L'opération Val_3 du composant A_3 retourne la valeur d'une variable d'état x_4 de A_4, consultée via le lien SEES. Cette version de Op_1 est un raffinement correcte de son abstraction dans A_1. En termes d'ordonnancement d'arbres et de modèles, on a donc $T_1 \leq T_2$ et $\Phi(T_1)$ ◊ $\Phi(T_2)$.

Soit enfin T_3 l'arbre correspondant à la totalité de l'architecture. Le dépliage va produire une nouvelle version de Op_1, car Op_2 a maintenant été raffinée dans AI_2. Or, cette version concrète de Op_2 appelle une opération qui modifie la variable x_4 : le code final de l'opération Op_1 retourne FALSE. On a donc un exemple d'architecture telle que tous les composants sont prouvés corrects, mais l'implémentation finale viole la spécification de départ. En termes d'ordonnancement d'arbres et de modèles, on peut établir $T_2 \leq T_3$ mais pas $\Phi(T_2)$ ◊ $\Phi(T_3)$.

Cet exemple met en évidence un problème d'aliasing pour la variable x_4. Au sein de AI_1, l'opération Op_2 est appelée dans un contexte où existe la variable x_4, fournie par un lien SEES. Cette variable ne fait pas partie du contexte de A_2, et la version abstraite de Op_2 ne peut donc pas la modifier. Malheureusement, la version concrète de Op_2 modifie x_4 : cette version concrète ne peut donc être utilisée pour raffiner les opérations de AI_1.

Sauf à interdire la clause SEES, on ne peut éviter l'apparition de phénomènes d'aliasing. On peut cependant éviter les effets de bord dus à l'aliasing, si on trouve une condition architecturale qui garantisse que les raffinements des opérations ne peuvent modifier les variables vues. Une telle condition architecturale a été proposée par [Rouzaud 1999], et nous la rappelons dans la Figure 3.5. Elle est obtenue par raisonnement sur les différents cas qui introduisent un lien SEES. Informellement, cette condition assure qu'on ne peut modifier, via un appel à une opération importée, l'état d'une machine vue (directement ou indirectement).

La condition suffisante sur l'architecture d'un développement pour assurer sa correction :

$$(uses\ ;\ can\_alter) \cap ((imports\ ;\ s^{+}) \cup (sees\ ;\ s^{*})) = \varnothing$$

s, *sees*, *imports*, *uses* et *can_alter* sont des relations définies entre les composants de la manière suivante :

1. $C_1\ s\ M_1$ ssi le composant C_1 (une MACHINE, un REFINEMENT ou une IMPLEMENTATION) voit (lien SEES) la MACHINE M_2
2. $M_1\ sees\ M_2$ ssi l'implémentation de M_1 voit la MACHINE M_2
3. $M_1\ imports\ M_2$ ssi l'implémentation de M_1 importe (lien IMPORTS) la MACHINE M_2
4. $M_1\ uses\ M_2$ ssi l'implémentation de M_1 voit ou importe la MACHINE M_2 :

   $uses = sees \cup imports$
5. M_1 *can_alter* M_2 ssi l'implémentation de M_1 peut modifier les variables de l'implémentation de M_2 : $can\_alter = (uses^{*}\ ;\ imports)$

Les opérateurs +, * et ; sont respectivement la fermeture transitive, la fermeture transitive et réflexive et la composition des relations.

Figure 3.5. Condition architecturale de [Rouzaud 1999]

3.3.3.2 Modèles aplatis issus de la machine racine de l'application

Supposons que l'architecture du projet B considéré satisfasse la condition de [Rouzaud 1999]. Soit M la machine racine du projet, et soient deux sous-arbres T_1, T_2 appartenant à $\mathcal{ST}$, et ayant pour racine M. A-t-on maintenant la propriété $T_1 \leq T_2 \Rightarrow \Phi(T_1)\ \Diamond\ \Phi(T_2)$? La réponse est négative, et la Figure 3.6 nous permet de donner deux contre-exemples.

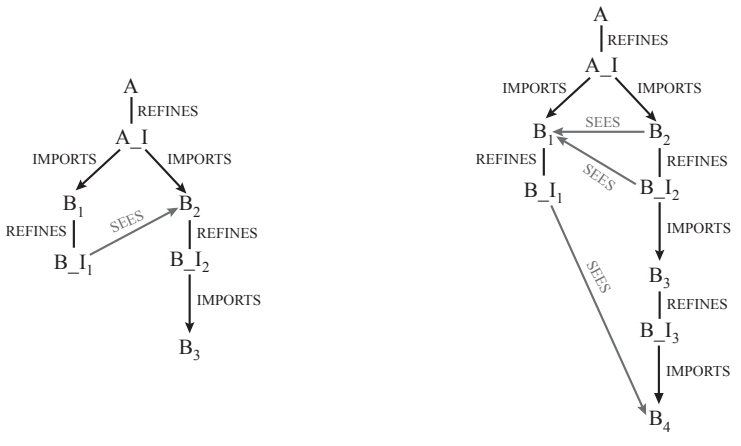


Figure 3.6. Exemples illustratifs

Pour la première architecture (a) qui satisfait la condition de [Rouzaud 1999], on peut obtenir, en cours de développement, le sous-arbre T_1 de racine A et de feuilles B_1, B_3. Dans $\Phi(T_1)$, les opérations de B_1 ne peuvent pas modifier les variables de B_3, qui font partie du contexte d'appel au sein de A_I. Cependant, lorsqu'on raffinera B_1 à l'étape suivante, on introduira un lien SEES qui rend ces variables indirectement modifiables depuis B_1. Il n'est donc pas garanti que $\Phi(T_1)$ constitue une abstraction de l'implémentation finale : le sous-arbre T_1 ne doit pas être pris comme étape de développement pour l'application. Notons que ce fait ne peut être établi qu'avec la connaissance des liens SEES qui seront introduits aux étapes ultérieures. Il semble alors nécessaire d'attendre la fin d'un développement pour en déterminer les étapes. Dans l'exemple, on pourrait décider a posteriori que T_1 ne constitue pas une étape de développement, mais que le sous-arbre de racine A et de feuilles B_I_1 et B_2 en constitue une.

Ceci nous amène à préconiser un usage plus restrictif du lien SEES. Il s'agit d'exclure a priori la possibilité d'introduire, entre deux étapes, des liens SEES qui remettent en cause le statut non-modifiable des variables contextuelles. Pour chaque composant ajouté à l'arbre de l'architecture, les nouveaux liens SEES introduits peuvent être caractérisés par la relation suivante :

$$\textit{nouveau-sees} = \textit{voit}^{+} - (\textit{père} ; \textit{voit}^{+})$$

où $\textit{père} = \textit{est\_raffiné\_par}^{-1} \cup \textit{importe}^{-1}$, et *voit*, *est_raffiné_par*, *importe* sont les relations issues du graphe $\mathcal{G}$ de l'architecture (voir le paragraphe 3.3.2). Par rapport aux définitions de la Figure 3.5, $\textit{voit} = s$ mais $\textit{importe} \neq \textit{imports}$, car *importe* exprime une relation entre un composant IMPLEMENTATION et un composant MACHINE. Nous allons maintenant renforcer la condition de [Rouzaud 1999] en ajoutant deux contraintes :

(C1) Tout nouveau lien SEES introduit par un composant MACHINE doit être strictement dans la portée d'un IMPORTS (comme le lien introduit par B_2 dans la Figure 3.6.b), ou alors doit pointer vers un composant qui ne sera pas raffiné.

(C2) Tout nouveau lien SEES introduit par un composant REFINEMENT ou IMPLEMENTATION doit pointer vers un composant qui ne sera pas raffiné.

Si ces contraintes ne sont pas respectées, et si un modèle intermédiaire est construit en cours de développement, alors le choix de ce modèle devra être validé a posteriori : d'abord en identifiant les liens SEES de l'architecture finale susceptibles de poser problème pour ce modèle, puis en analysant plus finement le contenu des opérations dans l'implémentation finale. Par exemple, dans le cas de l'architecture (a) de la Figure 3.6, le choix du sous-arbre T_1 était valide s'il s'avère que les opérations de B_I_1 ne modifient aucune des variables introduites dans B_I_2 et B_3.

Au contraire, l'usage restrictif du lien SEES que nous proposons permet l'identification d'étapes de développement sur une architecture partielle, sans avoir à prendre en compte les liens SEES ultérieurement introduits. Il reste cependant à déterminer plus précisément quels sont les sous-arbres que nous pouvons retenir a priori comme étapes, et pour cela nous allons nous appuyer sur l'exemple (b) de la

Figure 3.6. Pour cette architecture, qui satisfait la condition de [Rouzaud 1999] renforcée avec (C1) et (C2), on peut construire le sous-arbre T_1 de racine *A* et de feuilles *B_I*$_1$, B_2. Le modèle déplié correspondant, $\Phi(T_1)$, a un lien SEES vers le composant B_4. Or, ce lien introduit un problème d'aliasing. Dans $\Phi(T_1)$, les opérations sont plongées dans un contexte où les variables de B_4 sont supposées non-modifiables. Mais le composant B_4 sera importé par le bas, via des raffinements et décompositions de B_2. Les versions concrètes des opérations pourront donc modifier ses variables. Soit T_2 l'arbre correspondant à la totalité de l'architecture : on a $T_1 \leq T_2$ mais pas nécessairement $\Phi(T_1) \Diamond \Phi(T_2)$.

Une manière d'éviter ce problème est de ne considérer que les modèles aplatis qui ne comportent pas de clause SEES. La mise à plat du composant racine est évidemment une machine abstraite indépendante, de même que la mise à plat de l'architecture complète. Pour les modèles intermédiaires, on ne retient que les sous-arbres *T* de $\mathcal{ST}$ tels que tous les liens SEES peuvent être résolus à l'intérieur de *T* (toute MACHINE vue par un composant de *T* est importée par un autre composant de *T*). Ceci nous amène donc à la définition suivante :

*Une **étape de développement d'un projet** de racine M est caractérisée par l'obtention d'un sous-arbre T de racine M tel que* $T \in \mathcal{ST}$ *et* $\Phi(T)$ *ne contient pas de clause SEES.*

Dans l'exemple (b) précédent, si la mise à plat de l'arbre de racine *A* et de feuilles B_1, B_2 donne un modèle encore trop imprécis vis-à-vis des exigences fonctionnelles, on pourra s'intéresser à l'étape de développement suivante, caractérisée par l'arbre de racine *A* et de feuilles B_1, B_3.

On a maintenant la propriété désirée, pour tout projet qui respecte la condition de [Rouzaud 1999] renforcée avec (C1) et (C2) : si T_1 et T_2 caractérisent des *étapes de développement*, $T_1 \leq T_2 \Rightarrow \Phi(T_1) \Diamond \Phi(T_2)$.

3.3.3.3 Modèles aplatis issus d'un sous-système de l'application

La notion d'étape de développement, telle que définie dans le paragraphe précédent, peut être étendue à la notion d'étape de développement d'un sous-système indépendant. Cependant, le cas de sous-systèmes ayant des liens SEES vers d'autres sous-systèmes doit être discuté.

Précisons tout d'abord que nous ne considérerons pas le cas de sous-systèmes introduisant des dépendances croisées. Soient deux sous-systèmes SS_1 et SS_2 correspondant à des arborescences disjointes ; s'il existe un lien SEES d'un composant de SS_1 vers un composant de SS_2, et un lien SEES d'un composant de SS_2 vers un composant de SS_1, alors la notion d'étape de développement n'est définie ni pour SS_1, ni pour SS_2. Dans ce cas, nous pensons que les développements de ces sous-systèmes ne doivent pas être considérés indépendamment l'un de l'autre. De plus, pour éviter les problèmes d'aliasing, nous imposons que tous les composants vus par SS_1 (selon la relation s^+) appartiennent à des branches distinctes du développement : si un

composant de SS_1 voit un composant de SS_2, on est ainsi sûr de ne pas pouvoir faire référence à un ancêtre ou un descendant de ce dernier.

Ayant écarté ces cas, nous étendons la notion d'étape de développement comme suit :

*Une **étape de développement d'un sous-système** de racine M est caractérisée par l'obtention d'un sous-arbre T de racine M tel que $T \in \mathcal{ST}$ et $\Phi(T)$ ne référence, dans sa clause SEES, que des composants extérieurs à ce sous-système.*

Les modèles intermédiaires ainsi obtenus ne peuvent conduire à des problèmes d'aliasing : si T_1 et T_2 caractérisent des étapes de développement, $T_1 \leq T_2 \Rightarrow \Phi(T_1) \Diamond \Phi(T_2)$.

3.3.4 Dépliage de modèles B

Nous avons introduit, dans le paragraphe 3.3.2, les conditions architecturales qui définissent le domaine d'entrée $\mathcal{ST}$ de la fonction de dépliage. Nous présentons dans ce paragraphe l'algorithme de cette fonction proprement dit. Notons que tout sous-arbre de $\mathcal{ST}$ ne correspondra pas à une étape de développement, tel que défini dans le paragraphe 3.3.3.

L'algorithme de dépliage se base sur des mécanismes d'enrichissement correspondant aux liens existants en B. Nous présentons ces mécanismes dans le paragraphe 3.3.4.1. L'algorithme de dépliage fait l'objet du paragraphe 3.3.4.2. Cet algorithme a été proposé dans [Waeselynck et Behnia 1997]. Nous terminerons dans le paragraphe 3.3.4.3, par la présentation de quelques problèmes liés à l'implémentation du dépliage.

3.3.4.1 Mécanismes d'enrichissement

A partir d'un ensemble de composants B, liés par des liens REFINES, IMPORTS, INCLUDES, SEES et USES, l'algorithme de dépliage est chargé de générer le texte formel déplié correspondant. Nous avons défini des mécanismes d'enrichissement propres à chacun de ces liens. Pour définir ces mécanismes, nous nous sommes basés sur la description donnée dans le B-Book pour chaque lien.

Enrichissement des liens IMPORTS et INCLUDES

Le lien IMPORTS permet à un composant de type IMPLEMENTATION de se servir des données et des opérations des instances de MACHINEs importées pour implémenter ses propres données et opérations. Dans la Figure 3.7, nous montrons le mécanisme d'enrichissement du lien IMPORTS à l'aide des composants symboliques MA_n et MB. Ce mécanisme n'est pas explicitement décrit dans le B-Book, nous l'avons défini en nous basant sur la description informelle du lien IMPORTS.

Si l'IMPLEMENTATION MA_n importe la MACHINE MB, le composant déplié est un REFINEMENT qui est construit de la manière suivante :

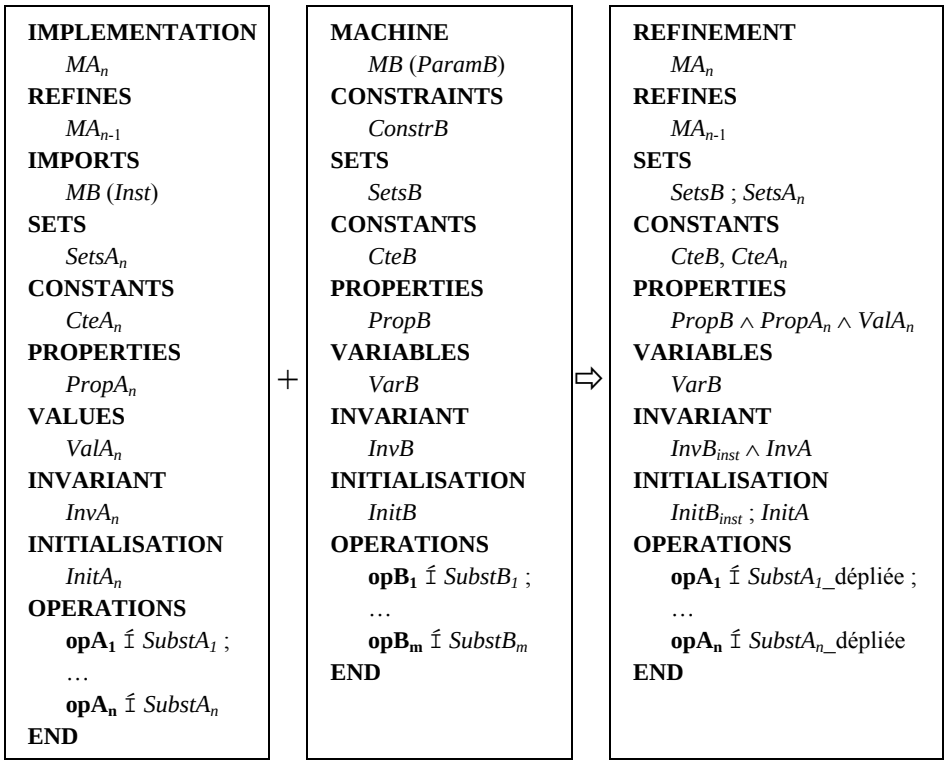


Figure 3.7. Mécanisme de base d'enrichissement du lien IMPORTS

- Le nom du composant est celui de MA_n, la clause REFINES de MA_n est recopiée dans le composant déplié.
- Dans les clauses INVARIANT, INITIALISATION et OPERATIONS de *MB*, les paramètres de *MB* sont remplacés par leur instanciation. Ceci revient à calculer, pour chaque clause, une substitution affectant aux paramètres formels leur valeur d'appel. Par exemple, pour la clause INVARIANT, il faut calculer la substitution :

$$[ParamB := Inst]\ InvB$$

- Le contenu des clauses SETS, CONSTANTS, PROPERTIES, INVARIANT et INITIALISATION de *MB* est concaténé avec celui de MA_n. Le contenu de la clause VALUES de l'IMPLEMENTATION MA_n est concaténé avec les propriétés *PropB* et $PropA_n$ dans le composant déplié. L'ordre dans lequel les deux composants sont initialisés est important. En effet, *MB* doit être initialisé avant MA_n, car les opérations de *MB* peuvent être appelées dans l'initialisation de MA_n. L'invariant de *MB* doit alors déjà avoir été établi.
- La clause VARIABLES de *MB* est recopiée dans le composant déplié.
- Les opérations du composant déplié sont celles de MA_n. Dans les opérations et l'initialisation du composant déplié, les appels aux opérations de *MB* sont

remplacés par leurs corps, les paramètres d'entrée et de sortie étant instanciés par les paramètres d'appel. Le corps instancié de l'opération $y \leftarrow$ op (x) ǐ S, qui est appelée par les paramètres d'appel a et b ($b \leftarrow$ op (a)) est obtenu par le calcul de la substitution $[y, x := b, a]$ $[ParamB := Inst]$ S.

Le lien INCLUDES permet de regrouper, dans une MACHINE ou un REFINEMENT, les données d'instances des MACHINEs incluses, ainsi que leurs propriétés, afin de créer un composant enrichi. Le mécanisme d'enrichissement de ce lien est globalement similaire à celui du lien IMPORTS. Si *MA* est une MACHINE qui inclut *MB*, le composant déplié est une MACHINE dont le nom, les paramètres formels et leur propriétés, décrites dans la clause constraints, sont ceux de MA. Les autres clauses du composant déplié sont construites à partir des clauses de MA et de MB de manière similaire au lien imports. Un tel mécanisme est également décrit dans le manuel de référence du langage B [Steria 1998]. Si MA est un composant de type REFINEMENT qui inclut la machine MB, le composant déplié dans ce cas sera un REFINEMENT et il portera le nom de MA. La clause REFINES de MA sera également recopiée dans le composant déplié. Il faut noter qu'à la différence du lien IMPORTS, le lien INCLUDES est un mécanisme d'enrichissement local. Ainsi, pour le dépliage de ce lien, on ne peut pas utiliser les raffinements des composants inclus à la place de leur abstraction.

Dans ces mécanismes, nous supposons qu'il n'y a pas de conflit de nom entre les identificateurs des composants qui sont dépliés. Cette règle est imposée par la méthode B pour éviter que, dans une clause d'un composant, il soit possible d'accéder à plusieurs constituants portant le même nom mais désignant des constituants différents, sans savoir lequel est effectivement utilisé. En effet, si un composant MA accède à une instance de machine MB, alors une donnée de MB accessible par MA ne doit pas porter le même nom qu'une donnée de MA.

Enrichissement du lien REFINES

Un composant de type REFINEMENT ne constitue pas une machine abstraite à part entière. Il contient le supplément qui doit être ajouté à un composant MACHINE afin de construire la machine abstraite correspondante. Cette dernière peut être construite de manière systématique en combinant les textes formels du composant MACHINE et de son REFINEMENT, comme nous l'avons vu dans le paragraphe 1.4.3.

Le mécanisme d'enrichissement du liens refines se base sur cette combinaison systématique. Nous le rappelons dans la Figure 3.8 pour deux composants symboliques *MA* et MA_1.

Le composant déplié est alors une MACHINE qui est construite de la manière suivante :

- Les paramètres formels du composant ainsi que leurs propriétés, décrites dans la clause CONSTRAINTS, sont ceux de *MA*.

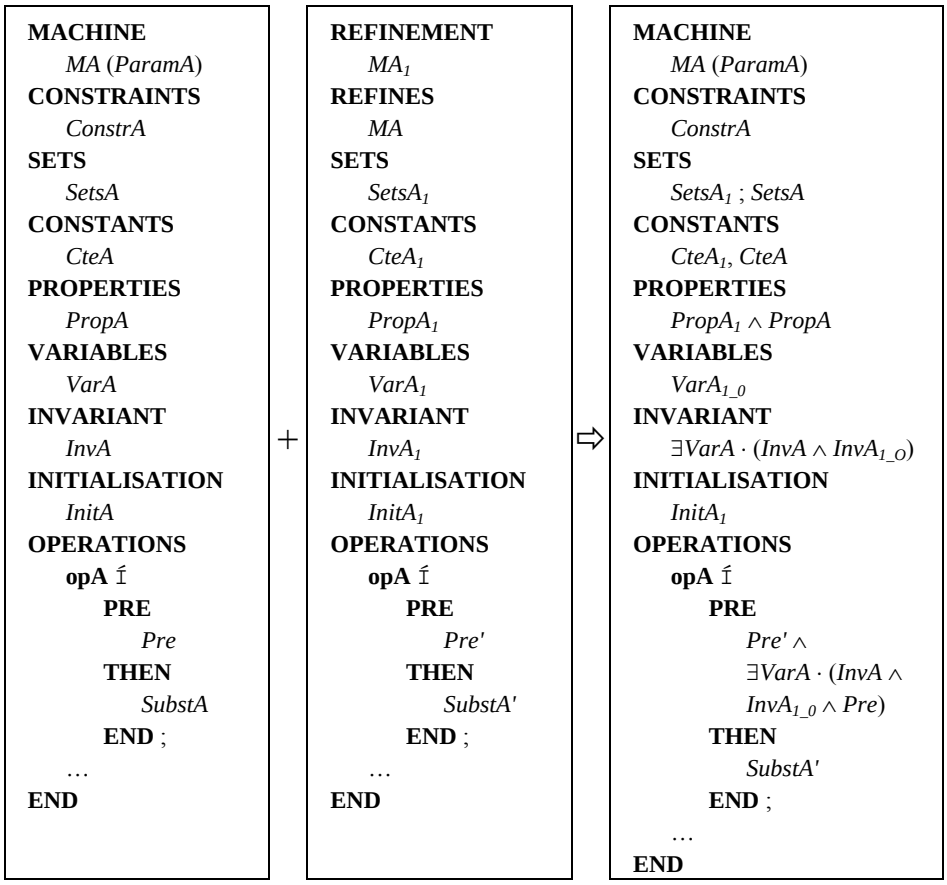


Figure 3.8. Mécanisme de base d'enrichissement du lien REFINES

- Dans le composant MA_1, si une variable, par exemple la variable *y*, a le même nom qu'une variable de *MA*, la variable *y* du composant MA_1 est renommée, par exemple à *y_0*, et un invariant de liaison $y = y\_0$ est ajouté à la clause INVARIANT de MA_1. Nous obtenons ainsi une nouvelle liste de variables $VarA_{1\_0}$ et un nouvel invariant $InvA_{1\_0}$ pour MA_1.
- Le contenu des clauses SETS, CONSTANTS et PROPERTIES de MA_1 est concaténé avec celui de *MA*. Les clauses VARIABLES et INITIALISATION de MA_1 sont recopiées dans le composant déplié.
- L'invariant et les opérations du composant déplié sont construits comme il est montré dans la Figure 3.8. Les opérations de *MA* qui ne sont pas raffinées dans MA_1 sont recopiées telles quelles dans le composant déplié.

Nous supposons que les composants *MA* et MA_1 ne contiennent plus de clause INCLUDES ou IMPORTS. Nous verrons que dans notre algorithme, les clauses INCLUDES ou IMPORTS sont d'abord traitées, en préliminaire à l'enrichissement selon REFINES.

Dans le cas d'une chaîne de liens REFINES (*MA, MA_1 , MA_2*), on déplie d'abord le lien entre *MA* et MA_1, puis le lien entre MA_2 et le composant déplié ainsi obtenu.

Liens de partage USES et SEES

Dans le B-Book, il n'existe pas de mécanisme explicite pour l'enrichissement des composants introduisant des liens USES et SEES. Ces liens se traduisent dans les obligations de preuves par un enrichissement du contexte. Nous ne proposons pas non plus de mécanisme d'enrichissement propre à ces liens. Ainsi, comme nous allons l'expliquer, les liens USES et SEES doivent être pris en compte respectivement au cours du dépliage des liens INCLUDES et IMPORTS.

Le lien USES permet uniquement une utilisation statique : il permet de partager un texte formel déclaratif entre plusieurs MACHINEs ; aucune opération de la MACHINE utilisée ne peut être appelée par les opérations des MACHINEs utilisatrices. Le lien USES ne permet pas d'instancier la MACHINE utilisée. Si une MACHINE *MA* utilise – par le lien USES – la MACHINE *MB*, elle doit être obligatoirement incluse en même temps que *MB* dans un autre composant *M*. L'instance de *MB* qui est réellement utilisée par *MA* est celle créée quand les deux MACHINEs sont incluses dans *M*. La résolution des références partagées doit alors être réalisée au moment de l'inclusion des deux MACHINEs dans le composant *M*, par le mécanisme d'enrichissement INCLUDES.

Le lien SEES permet d'établir des courts-circuits entre des branches indépendantes du développement. Un composant de type MACHINE, REFINEMENT ou IMPLEMENTATION peut voir par lien SEES un composant de type MACHINE afin de consulter les constituants de celui-ci sans les modifier. Une MACHINE vue doit obligatoirement être importée une fois dans le développement : quand un composant *MA* voit la MACHINE *MB*, l'instance qui est réellement vue est celle qui est créée au moment de l'importation de *MB* par un ancêtre commun de *MA* et *MB* dans le graphe de développement. Les références créées par le lien SEES doivent alors être résolues au moment de l'importation des deux composants dans leur ancêtre commun, par le mécanisme d'enrichissement IMPORTS.

Contrairement au lien USES, la résolution des références créées par le lien SEES peut ne pas être directe. L'ancêtre commun des deux composants liés par le lien SEES peut remonter plusieurs niveaux d'importation. Les liens SEES non-résolus doivent alors être propagés par les mécanismes d'enrichissement des liens IMPORTS, INCLUDES et REFINES. Ainsi, supposons qu'une IMPLEMENTATION *I* importe M_1 et M_2, le composant M_2 ayant un lien SEES M_3. Si *I* contient une instance de M_3, on procède à la résolution du lien SEES. Sinon, une clause SEES M_3 est ajoutée à *I*.

3.3.4.2 Algorithme du dépliage

Après avoir présenté les différents mécanismes d'enrichissement, nous allons maintenant proposer un algorithme de mise à plat de modèles.

Rappelons tout d'abord quelques éléments que nous avons présentés dans le paragraphe 3.3.2. A partir du graphe représentant l'architecture du projet B, une

structure d'arbre $\mathcal{T}$ peut être extraite. Cette structure d'arbre est une vue simplifiée du graphe complet et ne contient explicitement que les liens REFINES et IMPORTS. Elle permet de distinguer deux catégories de composants : 1) les nœuds qu'elle comporte, et 2) les composants auxiliaires qui sont transitivement inclus par chacun de ses nœuds. Nous avons ensuite défini le domaine d'entrée valide de la fonction de dépliage, Φ, par un ensemble $\mathcal{ST}$ des sous-arbres de $\mathcal{T}$. Les éléments de $\mathcal{ST}$ vérifient les conditions architecturales nécessaires à l'obtention d'un texte formel déplié.

Pour faciliter la présentation de l'algorithme, nous utilisons des chaînes de caractères dénotant des sous-arbres. Ces chaînes sont construites à partir des noms des composants et des symboles « /.[,] » :

- $\boldsymbol{N_1 / N_2}$ signifie qu'il existe un arc étiqueté « est_raffiné_par » entre les nœuds N_1 et N_2, et
- $\boldsymbol{N.[T_1, \ldots, T_n]}$ signifie qu'il existe un arc étiqueté « importe » du nœud N vers chaque sous-arbre T_1, …, T_n.

De plus, nous désignons par $"N'$ le texte formel du composant N.

L'algorithme de dépliage est basé sur trois fonctions, correspondant au dépliage des liens INCLUDES, IMPORTS et REFINES selon les principes présentés dans le paragraphe 3.3.4.1 pour chacun de ces liens. Ces fonctions sont :

- enrich_inc ($"N'$), retournant le texte formel construit à partir du texte formel $"N'$ en faisant la fermeture transitive des liens INCLUDES ($"N'$ est inchangé s'il ne contient pas de clause INCLUDES).
- enrich_imp ($"N'$, $"N_1'$, …,$"N_m'$), retournant le texte formel construit à partir de $"N'$, qui contient une clause IMPORTS référençant m composants, et de $"N_1'$, …,$"N_m'$. Les textes formels importés $"N_1'$, …,$"N_m'$ peuvent correspondre à des textes formels générés au cours du dépliage.
- enrich_ref ($"N'$, $"N_1'$), retournant le texte formel construit à partir de $"N'$, décrivant un composant MACHINE, et de $"N_1'$ décrivant un REFINEMENT de N. De même que précédemment, $"N'$ et $"N_1'$ peuvent être des textes formels générés au cours du dépliage.

Nous pouvons maintenant donner l'algorithme de dépliage. Le programme Φ parcourt tout élément $T \in \mathcal{ST}$ comme suit :

- si $T = N$, alors $\Phi(T)$ = enrich_inc ($"N'$).
- si $T = N_1/\ldots/N_m$ il est possible de décomposer T en deux sous-arbres T' et T'' tels que $T' = N_1/\ldots/N_{m-1}$ et $T'' = N_m$, Alors, $\Phi(T)$ = enrich_ref ($\Phi(T')$, $\Phi(T'')$).
- de façon similaire, $T = N_1/\ldots/N_m.[T_1,\ldots,T_n]$ est décomposé en $T' = N_1/\ldots/N_{m-1}$ et $T'' = N_m.[T_1,\ldots,T_n]$. Alors, $\Phi(T)$ = enrich_ref ($\Phi(T')$, $\Phi(T'')$).
- si $T = N.[T_1,\ldots,T_n]$, alors $\Phi(T)$ = enrich_imp ($\Phi("N')$, $\Phi(T_1)$,…, $\Phi(T_n)$).

3.3.4.3 Problèmes liés à l'implémentation de l'algorithme du dépliage

L'algorithme de dépliage décrit dans le paragraphe précédent est une description abstraite du traitement réel à effectuer. En effet, l'implémentation d'un prototype partiel de la fonction de dépliage a mis en évidence la nécessité d'effectuer de nombreux contrôles et pré-traitements, au cours du dépliage. Dans ce paragraphe, nous évoquons les problèmes identifiés et les solutions qui peuvent être envisagées.

Conflits de noms

Dans une architecture B, l'absence de conflit de noms entre les identificateurs des composants est exigée. Ce contrôle est effectué de manière locale par les outils B. Par exemple, si une MACHINE M_A voit (lien SEES) une autre MACHINE M_B, l'Atelier B vérifie que les noms des paramètres formels de M_A, les noms de ses ensembles (abstraits et énumérés), de ses éléments énumérés, de ses constantes et de ses variables d'état soient distincts de tous les noms utilisés dans M_B. Cette absence de conflit de noms n'est plus garantie pour les composants intermédiaires générés au cours du dépliage : si M_B est un composant déplié, il peut contenir des données qui n'étaient pas accessibles localement. Par exemple, une donnée abstraite peut être remplacée par une donnée concrète. Des conflits de nom peuvent alors exister et doivent être résolus par renommage. De plus, si une donnée d'un composant est renommée, ce renommage doit être propagé à tout composant pouvant accéder à la donnée renommée.

Des conflits de nom peuvent également exister au moment de l'expansion du corps des opérations, entre les paramètres d'appel et les variables locales de l'opération appelée. En effet, un paramètre d'appel peut porter le même nom qu'une variable locale définie dans le corps de l'opération appelée. Considérons par exemple l'opération suivante :

$v \leftarrow$ op (i) ΐ VAR w IN
$w := i * 2$; $v :=$ bool ($w \neq$ MAXINT)
END

Un appel à cette opération peut être $z \leftarrow$ op (w). Lors de l'expansion de cet appel à l'opération, un conflit de nom existera entre le paramètre d'appel et la variable locale définie dans le corps de l'opération. L'un de ces identificateurs doit alors être renommé. Le renommage de la variable locale nécessitant d'effectuer moins de modifications, c'est la solution que nous avons choisie lors de l'implémentation du prototype partiel.

Le problème de renommage des identificateurs est un problème classique que nous avons dû étudier dans le cadre des architectures B. Lors de l'implémentation du prototype du dépliage, nous avons choisi de gérer le renommage des identificateurs en cours de traitement, et de propager ce renommage vers tout composant accédant à la donnée renommée. Ce choix s'est avéré lourd à mettre en œuvre, tant pour la détection en-ligne de conflits que pour la propagation du renommage. La solution que nous préconisons maintenant est d'effectuer en pré-traitement un renommage systématique de tous les identificateurs afin d'exclure a priori la possibilité de

conflits. Par exemple, tous les identificateurs (les paramètres formels, les constantes, les ensembles, les éléments des ensembles énumérés, les variables d'état, les paramètres d'entrée et de sortie des opérations et les variables locales) de tous les composants peuvent être renommés en préfixant leur nom par leur chemin d'accès. Le chemin d'accès d'un composant commence par la machine abstraite de plus haut niveau dans le développement et suit les liens REFINES, IMPORTS et INCLUDES. Ainsi, si la MACHINE de plus haut niveau est *M*, raffinée par l'IMPLEMENTATION *M_I*, qui importe les machines *M_A* et *M_B*, l'identificateur *x* du composant *M_A* est renommé par *M.M_I.M_A.x*.

Un tel renommage systématique permet de garantir que tous les identificateurs de tous les composants portent un nom unique, évitant ainsi d'avoir à traiter les conflits.

Résolution du lien SEES

Comme nous l'avons mentionné dans le paragraphe 3.3.4.1, nous n'avons pas défini de mécanismes d'enrichissement propres aux liens SEES et USES. Les références créées par ces liens sont mises à plat respectivement lors du dépliage des liens IMPORTS et INCLUDES. Cependant, la résolution des références dues au lien SEES s'avère plus difficile car, contrairement au lien USES, elle peut ne pas être directe. Les informations concernant ces références doivent alors être conservées et propagées au cours du dépliage.

Considérons l'exemple de la Figure 3.6.b, que nous avons utilisé dans le paragraphe 3.3.3.2. La MACHINE B_2 voit la MACHINE B_1. Les références créées par ce lien SEES sont résolues quand les deux composants sont importés dans leur ancêtre commun, le composant *A_I* dans notre exemple. La résolution de ces références tient compte de l'instanciation des paramètres formels du composant vu. Par exemple, si le composant B_1 est paramétré, ces paramètres peuvent être utilisés dans le corps des opérations de lecture appelées par B_2. Ces paramètres sont instanciés quand B_1 est importé dans *A_I*.

Cependant, l'ancêtre commun de deux composants ayant un lien SEES peut remonter dans le graphe de développement, comme c'est le cas pour l'ancêtre commun des composants B_I_1 et B_4 dans notre exemple. Lors du dépliage du lien d'importation entre B_I_3 et B_4, nous allons ajouter le texte du composant B_4 à celui du composant B_I_3. Mais le texte formel du composant B_4 ne doit pas être détruit, comme c'était le cas dans une première implémentation du prototype partiel de la fonction du dépliage. Nous devons conserver une copie des opérations de B_4 qui sont appelées par les opérations de B_I_1, ainsi que l'instanciation de ses paramètres formels éventuels. Ces informations seront utilisées par *A_I*, pour résoudre les références non-résolues en provenance de B_I_1.

Un autre problème lié au lien SEES est la détermination de l'ordre des initialisations. Un composant qui est importé ou vu par un autre composant doit être initialisé avant ce dernier. L'ordre des initialisations ne peut alors être déterminé qu'au niveau du composant racine de l'architecture mise à plat, car cela nécessite de prendre en compte toutes les dépendances entre composants, et en particulier les courts-circuits entre branches du développement introduits par les liens SEES.

Pour déterminer l'ordre des initialisations, un algorithme utilisé par l'Atelier B pour la génération du code source, est le suivant :

- A chaque composant est associé un booléen. La valeur de ce booléen est vraie si le composant a été déjà initialisé.
- Lors du parcours de l'arbre des IMPORTS, l'initialisation de chaque composant se décline comme suit :

  Si le composant n'est pas initialisé :

  - les composants importés sont initialisés dans l'ordre de déclaration ;
  - les composants vus sont initialisés dans l'ordre de déclaration ;
  - le composant est initialisé.

Nous proposons de déterminer l'ordre des initialisations, avant d'effectuer le dépliage, en utilisant cet algorithme. Ainsi, pour l'exemple de la Figure 3.6.b, nous pouvons déterminer l'ordre suivant :

1. B_4 (parcours du chemin $A \rightarrow B_1 \rightarrow B_4$)
2. B_1 (remontée dans le chemin $A \rightarrow B_1$)
3. B_3 (parcours du chemin $A \rightarrow B_2 \rightarrow B_3$, sachant que B_4 est déjà initialisé)
4. B_2 (remontée dans le chemin $A \rightarrow B_2$)
5. A (remontée dans le chemin A)

Problème de fusion des états

En plus des problèmes que nous venons de décrire, il faut s'intéresser au problème particulier de la substitution skip. En effet par définition, cette substitution porte sur les variables d'état présentes dans le composant : elle spécifie que ces variables restent inchangées. Or, lors du dépliage, le composant nouvellement créé incorpore de nouvelles variables d'état provenant des composants dépliés. Nous sommes alors confrontés au problème de la portée de la substitution skip : quelles sont les variables d'état concernées et quelles sont celles qui ne le sont pas ? Le problème se pose lorsque skip est mise en parallèle avec une autre substitution : les variables concernées doivent être disjointes.

Pour résoudre ce problème, deux solutions éventuelles peuvent être envisagées :

- Une nouvelle substitution skip_x peut être définie, qui indique l'ensemble des variables d'état x concernées.
- La substitution skip peut être remplacée par une substitution équivalente qui ne fait *rien*, par exemple par la substitution VAR x IN $x := 1$END.

Structure d'accueil

Suivant les points que nous venons de mentionner, on peut dégager du processus général de dépliage certaines opérations de base, qu'il serait intéressant de factoriser.

Lors du dépliage, nous avons souvent recours au renommage ou à l'instanciation des paramètres. Ce processus fait appel à la même opération qui doit remplacer un

identificateur par un autre identificateur ou expression, dans une substitution, une expression ou un prédicat donné. Par exemple, si l'identificateur *x* d'un composant est renommé en *x_1*, ceci revient à calculer la substitution *x* := *x_1* pour tous les prédicats, les expressions et les substitutions du composant. Ainsi, ce remplacement n'est pas un simple « rechercher / remplacer » syntaxique, mais doit tenir compte de la sémantique de la structure dans laquelle s'effectue le remplacement. Pour cela, les règles de substitution définies dans l'annexe E.2 du B-Book doivent être implémentées. Ces règles décrivent le calcul d'une substitution *x* := *E*, où *x* est une variable et *E* une expression, appliquée à un prédicat, une expression ou une substitution généralisée.

L'algorithme de dépliage que nous avons défini s'appuie sur deux types de structures de données : une structure représentant l'architecture arborescente du sous-ensemble du développement à déplier, et pour chaque composant, une structure de données représentant son source B. La construction pratique de ces structures de données requiert l'utilisation d'un parseur. Cet outil doit être capable d'analyser les fichiers B et d'identifier les différents liens (REFINES, IMPORTS, SEES, …) existant entre les composants. Ainsi, une représentation arborescente de l'architecture peut être construite. Si un composant est renommé lors d'une importation ou d'une inclusion, une copie explicite de celui-ci est ajoutée à l'arbre. Le parseur doit également permettre de construire, pour chaque composant, une structure de données permettant d'accéder directement à chacun de ses constituants. Par exemple, lors de l'enrichissement des liens, nous avons besoin d'avoir un accès direct au contenu des clauses dans chaque composant, ou plus finement au corps des opérations, à leur variables locales, etc.

Lors de l'implémentation du prototype de la fonction de dépliage, nous nous sommes basés sur la sortie d'un parseur développé par nos collègues Dorian Petit et Georges Mariano à l'INRETS. Ce parseur génère, pour chaque composant B, un arbre syntaxique stocké sous le format textuel XML. Nous utilisions ces fichiers texte en entrée du prototype de la fonction de dépliage. La manipulation des fichiers XML s'est avérée fastidieuse : nous devions les parser pour en extraire les informations nécessaires au dépliage. Bien que parser un fichier XML soit plus facile que parser un source B, cela nécessitait néanmoins le développement d'un outil supplémentaire. De plus, en ce qui concerne la représentation des constituants d'un composant B, la structure de données dont nous avons besoin étant déjà créée par le parseur pour ses traitements internes, il paraît inutile de la recréer à partir des fichiers XML. La solution maintenant envisagée est de développer un module englobant à la fois le parseur et la fonction de dépliage. De cette manière, la fonction de dépliage peut se servir de la structure interne du parseur, représentant l'arbre syntaxique des composants B. L'implémentation d'un tel outil de dépliage est en cours à l'INRETS.

3.4 Séquence de test et oracle

Nous avons présenté jusqu'ici le premier aspect de notre cadre unifié de test, c'est-à-dire comment identifier, dans un développement B, des modèles dont le comportement peut être testé. Ces modèles correspondent à des machines abstraites dépliées, construites à partir d'ensembles de composants du développement. L'identification de ces ensembles de composants se fait selon les conditions architecturales présentées dans les paragraphes 3.3.2 et 3.3.3. Nous nous intéressons maintenant au deuxième aspect de notre cadre unifié, c'est-à-dire la définition des notions de séquence de test et d'oracle pour les modèles ainsi obtenus aux différentes étapes de développement.

L'idée principale est de définir la notion d'oracle de test en termes d'une relation de raffinement. Comme nous l'avons expliqué dans le paragraphe 1.4, le raffinement des machines abstraites a été défini dans le B-Book à l'aide de substitutions externes. Une substitution externe contient une séquence finie d'appels aux opérations d'une machine abstraite, les paramètres d'entrée des opérations étant instanciés par des valeurs, et les paramètres de sortie par des variables d'appel distinctes. Du fait de l'encapsulation de l'état interne, tout ce que la substitution externe peut observer est l'établissement de post-conditions sur les paramètres de sortie des opérations.

Nous définissons une séquence de test comme suit :

Définition. Une séquence de test T est une substitution externe pouvant être implémentée sur la machine abstraite sous test M.

Cette définition de séquence de test est similaire à celle de [Alnet 1996]. Cependant, dans notre cas, la machine abstraite M est la machine abstraite dépliée correspondant au modèle qui doit être testé. De plus, comme notre objectif est de valider la spécification, les résultats attendus de la séquence de test doivent être déterminés à partir du cahier des charges. Nous supposons que ces résultats peuvent être spécifiés de la manière suivante :

Définition. Les résultats attendus sont exprimés par une autre substitution T_0 portant sur les mêmes paramètres de sortie que ceux observés par la séquence de test T.

La sémantique de T_0 détermine les post-conditions qui doivent être établies par les valeurs des paramètres de sortie calculées par les opérations de M. Si les besoins exprimés dans le cahier des charges sont déterministes, T_0 peut être une séquence de substitutions simples qui donne une valeur à chaque paramètre de sortie. Si les besoins indiquent plusieurs résultats possibles, correspondant à des choix d'implémentation laissés ouverts, T_0 peut exprimer un choix indéterministe sur un ensemble de valeurs.

Prenons l'exemple de la machine abstraite Little_Example_1, introduite dans le chapitre 1, à laquelle nous avons ajouté une opération give_element qui renvoie de manière indéterministe une des valeurs stockées dans l'ensemble y.

MACHINE Little_Example_1
VARIABLES y
INVARIANT $y \in \mathbb{F}(\mathbf{NAT_1})$
INITIALISATION
$y := \emptyset$
OPERATIONS
enter (n) = **PRE** $n \in \mathbf{NAT_1}$ **THEN** $y := y \cup \{n\}$ **END** ;
$m \leftarrow$ **maximum** = **PRE** $y \neq \emptyset$ **THEN** $m :=$ **max** (y) **END**
$n \leftarrow$ **give_element** = $n :\in y$
END

Une séquence de test T pour cette machine abstraite peut être :

T = BEGIN
enter (*1*);
enter (*2*);
$a \leftarrow$ give_element
END

Le résultat attendu de cette séquence de test peut être T_0 = $a :\in \{1, 2\}$, exprimant le fait que le résultat calculé par T est correct s'il appartient à l'ensemble $\{1, 2\}$.

A partir de ces définitions de T et T_0, nous allons maintenant spécifier comment l'oracle de test doit comparer les résultats attendus avec ceux calculés par le modèle sous test. Nous désignons par T_M l'implémentation de la séquence de test T sur la machine abstraite M. Rappelons que l'implémentation d'une substitution externe procède comme suit : M est initialisée et les appels aux opérations sont expansés par leur corps, les paramètres formels étant remplacés par les paramètres d'appels. Pour que l'oracle puisse comparer les substitutions T_0 et T_M, il faut que ces dernières portent sur le même ensemble de variables. Pour ce faire, les variables d'état de la machine abstraite M, présentes dans T_M, doivent être cachées en utilisant la substitution choix non-borné. L'état de M étant désigné par $v_M \in E_M$, nous définissons l'oracle de test comme suit :

Définition. L'oracle ne renvoie un verdict d'acceptation que si la relation de raffinement suivante est vérifiée : $T_0 \lozenge @v_M \cdot (v_M \in E_M \Rightarrow T_M)$

Nous allons maintenant montrer les conséquences de cette définition d'oracle pour le test d'un raffinement de M. Pour alléger la rédaction, nous adoptons le raccourci $@T_M$ pour désigner la substitution $@v_M \cdot (v_M \in E_M \Rightarrow T_M)$ obtenue en cachant les variables d'état de M.

Nous rappelons la définition du raffinement des machines abstraites, que nous avons expliquée dans le paragraphe 1.4.2 :

$$M \lozenge N \quad \Leftrightarrow \quad @T_M \lozenge @T_N \qquad \text{pour tout } T$$

Grâce d'une part à cette définition et d'autre part à la transitivité de la relation $\lozenge$, nous pouvons établir que si une machine abstraite fournit des résultats acceptables en

réponse à une séquence de test T, alors des raffinements prouvés de cette machine fourniront également des résultats acceptables en réponse à T :

$$T_0 \lozenge @T_M \quad \wedge \quad @T_M \lozenge @T_N \quad \Rightarrow \quad T_0 \lozenge @T_N \qquad (1)$$

Lors du test de la machine M, les résultats d'une séquence T seront refusés par l'oracle dans les deux cas suivants :

- Une incohérence est observée entre T_0 et T_M :

  $$\neg(T_0 \lozenge @T_M) \quad \wedge \quad \neg(@T_M \lozenge T_0)$$

  ce qui correspond à une faute dans la spécification. Dans notre exemple, si $@T_M = a := 3$, ce résultat incorrect sera refusé par l'oracle, car :

  $$\neg(a :\in \{1, 2\} \lozenge a := 3) \quad \wedge \quad \neg(a := 3 \lozenge a :\in \{1, 2\})$$

- Les résultats obtenus de T_M sont plus indéterministes que ceux exprimés par T_0 :

  $$@T_M \quad \lozenge \quad T_0$$

  Dans ce cas, des raffinements corrects de M peuvent ou non satisfaire les besoins, selon les choix pris par les développeurs. Un résultat de ce type dans notre exemple peut être : $@T_M = a :\in \{1, 2, 3\}$.

D'après (1), en ce qui concerne la séquence de test T, il existe une étape de développement à partir de laquelle le test des modèles formels n'apportera aucune information nouvelle. Cette étape de développement peut être située n'importe où dans le développement, entre la machine de plus haut niveau et le modèle final contenant toutes les IMPLEMENTATIONS, selon l'approche de conception B adoptée. Notons que nous avons défini l'oracle par une relation de raffinement et non par une relation d'équivalence, car le modèle testé peut être plus déterministe que les besoins, dans le cas où les besoins sont imprécis et laissent le choix aux concepteurs. Dans notre exemple, $@T_M = a := 2$ est un résultat qui sera accepté par l'oracle.

3.5 Conséquences du cadre unifié

Nous avons vu dans le paragraphe précédent que notre cadre unifié définit le test des machines abstraites dépliées de la manière suivante : une séquence de test est une substitution externe qui doit être implémentée sur la machine ; l'oracle de test vérifie si l'implémentation de la séquence de test raffine une autre substitution affectant des valeurs attendues (ou des ensembles de valeurs acceptables) aux variables de sortie. Nous allons discuter dans ce paragraphe des conséquences pratiques de ces définitions.

3.5.1 Implémentation de l'oracle via les obligations de preuve

L'oracle de test étant défini par une relation de raffinement entre deux substitutions, le test des modèles formels peut théoriquement s'effectuer en utilisant uniquement les obligations de preuve générées par la méthode B. A condition d'avoir un outil de

dépliage de modèles qui construit la machine abstraite équivalente à un ensemble de composants B, nous pouvons procéder de la manière montrée dans la Figure 3.9 pour implémenter les séquences de test. La machine Moniteur_de_Test est un composant B dont l'opération spécifie la substitution T_0, décrivant les résultats attendus des variables observables o_1, …, o_n. Le composant Moniteur_de_Test est raffiné par le composant Moniteur_de_Test_1 qui importe la machine abstraite aplatie sous test et dont l'opération spécifie la séquence de test *T*, portant sur les mêmes variables observables.

MACHINE *Moniteur_de_Test* **OPERATIONS** $o_1, \ldots, o_n \leftarrow$ **séquence_de_test** Í T_0 **END**	**IMPLEMENTATION** *Moniteur_de_Test_1* **REFINES** *Moniteur_de_Test* **IMPORTS** *Machine_Abstraite_Testée* **OPERATIONS** $o_1, \ldots, o_n \leftarrow$ **séquence_de_test** Í T **END**

Figure 3.9. Implémentation de l'oracle par les obligations de preuve B

Le lien IMPORTS implémente la séquence de test sur la machine abstraite testée. Les obligations de preuve de raffinement générées par la méthode B constituent alors l'oracle de test. Les règles de visibilité du lien IMPORTS garantissent l'encapsulation des variables d'état du composant importé. L'opération du Moniteur_de_Test_1 ne peut donc commander et observer que les paramètres d'entrée et de sortie des opérations de la machine abstraite importée. Les variables d'état concrètes (clause CONCRETE_VARIABLES) font cependant exception à cette règle. Les règles de visibilité du lien IMPORTS permettent que les variables concrètes soient directement consultées dans les opérations du composant qui les importe. On peut donc faire référence à ces variables concrètes dans la séquence de test *T* et vérifier l'établissement des post-conditions sur ces variables par rapport aux résultats attendus spécifiés dans T_0. Cette situation ne constitue pas un problème vis-à-vis du test des modèles abstraits : les variables concrètes sont garanties exister dans tous les raffinements de la machine abstraite testée et les obligations de preuve liées aux invariants de collage permettent d'assurer la conservation des post-conditions établies.

Si la machine abstraite sous test correspond à un modèle aplati issu d'un sous-système de l'application, elle peut contenir des liens SEES vers des composants extérieurs à ce sous-système. Dans ce cas, le moniteur de test doit fournir des bouchons (*stub*) pour chaque composant vu. Ainsi, l'IMPLEMENTATION Moniteur_de_Test_1 doit importer tous les composants nécessaires à Machine_Abstraite_Testée.

La particularité de l'approche d'implémentation de l'oracle par obligations de preuve est le fait de pouvoir exprimer le test dans le même formalisme que le modèle

sous test. Cependant, elle peut être très coûteuse et peu pratique. La machine abstraite aplatie sous test correspondrait à un ensemble de composants B qui contiendrait plusieurs niveaux de raffinement et de décomposition. Elle serait complexe et les obligations de preuve générées seraient difficiles à prouver.

3.5.2 Implémentation de l'oracle via l'animation

La méthode classique de test consiste à obtenir les résultats d'une séquence de test par exécution de code et à vérifier que chaque valeur obtenue est acceptable. Dans ce cas, l'oracle peut être un programme qui vérifie l'égalité des valeurs de sortie observées et attendues, ou l'appartenance d'une valeur observée à un ensemble de valeurs acceptables. Si le modèle B sous test est un modèle abstrait, obtenu à une étape intermédiaire du développement, les résultats doivent être calculés par animation de modèle.

Dans le cas d'utilisation d'un outil d'animation, le cadre unifié suppose que les résultats calculés par l'outil sont corrects vis-à-vis de la sémantique des substitutions généralisées. Nous avons considéré trois aspects des substitutions généralisées, c'est-à-dire l'indéterminisme, la terminaison et la faisabilité, et avons identifié les propriétés souhaitées de l'outil d'animation. Nous discutons de chacun de ces aspects dans les paragraphes suivants. Nous verrons ensuite si les outils d'animation des ateliers commerciaux de développement en B possèdent ces propriétés.

3.5.2.1 Indéterminisme

Le modèle sous test étant obtenu à une étape intermédiaire de développement, il peut contenir des substitutions indéterministes. Pour ces substitutions, l'outil d'animation doit produire l'ensemble des valeurs de sortie possibles.

Prenons l'exemple de la machine TRIANGLE appartenant au développement B de l'exemple du Triangle que nous avons présenté dans le chapitre 2, avec la séquence de test :

$$T \text{ Í } oo \leftarrow \text{Est\_un\_Triangle } (1, 2, 2)$$

et le résultat attendu T_0 Í oo := TRUE. L'implémentation de cette séquence de test sur cette machine résulte en la substitution suivante (voir la Figure 2.7.d) :

$T_{TRIANGLE}$ Í PRE
$1 \in \text{NAT} \wedge 2 \in \text{NAT} \wedge 2 \in \text{NAT} \wedge 1 \leq Max\_Nat \wedge$
$2 \leq Max\_Nat \wedge 2 \leq Max\_Nat \wedge 1 \leq 2 \wedge 2 \leq 2$
THEN
ANY b_t WHERE $b\_t \in \text{BOOL} \wedge$
$(b\_t = \text{TRUE} \Rightarrow 2 > 2 - 1)$
THEN $oo := b\_t$
END
END

D'après la sémantique des substitutions généralisées, à la fois TRUE et FALSE sont des valeurs possibles de *oo*. La substitution T_{TRIANGLE} n'est alors pas un raffinement de T_0. Si l'outil d'animation gère l'indéterminisme en calculant une seule valeur (par exemple TRUE), ce résultat partiel peut à tort être accepté par l'oracle.

Le résultat d'animation des substitutions indéterministes doit être dans une des catégories suivantes :

- L'outil d'animation n'est pas capable de calculer de résultats. Dans ce cas le test n'est pas concluant.
- L'outil calcule un sous-ensemble des résultats : il doit informer l'oracle que ce résultat est partiel. Dans ce cas le test n'est pas concluant si l'ensemble des résultats calculés est inclus dans l'ensemble des résultats attendus. Par exemple, pour la séquence de test précédente, si l'outil calcule le résultat partiel FALSE, l'oracle pourra rejeter ce résultat car quel que soit le résultat exact, il n'est pas un raffinement de T_0 Í *oo* := TRUE. Mais si l'outil calcule le résultat partiel TRUE, l'oracle ne peut ni accepter ni rejeter ce résultat, car le résultat exact peut être ou ne pas être un raffinement de T_0.
- L'outil calcule l'ensemble des valeurs de sortie. Dans ce cas l'oracle peut vérifier que chaque valeur calculée est incluse dans l'ensemble des valeurs attendues.

3.5.2.2 Terminaison

Si la séquence de test est dérivée d'analyses fonctionnelles indépendantes du développement B, il est possible qu'une opération soit appelée en dehors de sa pré-condition la plus externe. Ceci révèle une faute dans la spécification car les opérations spécifiées doivent se terminer au moins dans tous les cas exprimés par le cahier des charges. L'outil d'animation doit prendre en compte cette situation et échouer à fournir un résultat de sortie. Notons que l'action d'une substitution qui ne se termine pas peut être mal formée. Par exemple, considérons l'exemple suivant d'une opération utilisant une fonction partielle *f* :

$$op \text{ Í PRE } x \in \text{dom}(f) \text{ THEN } y := f(x) \text{ END}$$

Si l'opération est appelée en dehors de sa pré-condition, la valeur d'appel de *x* n'appartient pas au domaine de la fonction *f*. Dans ce cas, la partie action de la fonction, $y := f(x)$, est dépourvue de sens, et l'outil ne doit pas calculer de valeur pour *y*.

3.5.2.3 Faisabilité

La méthode B ne génère pas d'obligations de preuve de faisabilité car ce type d'obligations de preuve est remplacé par des preuves de construction. Un composant de type MACHINE ou REFINEMENT peut contenir des substitutions *miraculeuses* ou *infaisables*, mais si le projet se termine par un ensemble de composants IMPLEMENTATIONs prouvés, ceci n'a pas été le cas.

Si le modèle sous test est obtenu à une étape intermédiaire du développement, il peut contenir des substitutions miraculeuses. Supposons que la séquence de test *T* soit implémentée sur la machine abstraite *M*. Si T_M est infaisable, elle est le raffinement correct de toute substitution T_0 portant sur les mêmes variables observables. Dans ce cas, si le test du modèle formel est effectué par les obligations de preuve de raffinement (voir le paragraphe 3.5.1), la machine abstraite *M* sera toujours validée. Cependant, par définition, les substitutions infaisables ne sont pas implémentables : il n'existe aucune interprétation exécutable correcte de T_M qui fournirait des résultats de sortie. L'animation doit soit ne pas se terminer, soit annoncer qu'aucun résultat de sortie n'est calculable. Aucun de ces résultats ne permettra à l'oracle de conclure que les résultats calculés sont conformes aux besoins. Dans ce sens, le verdict obtenu pour le test par animation de modèle sera différent du verdict obtenu par le test via les obligations de preuve.

Notons que l'existence d'une substitution miraculeuse serait de toute façon détectée par les obligations de preuve de raffinement dans une étape ultérieure de développement puisque, comme nous l'avons rappelé, une substitution miraculeuse ne peut pas être implémentée. Cependant, l'animation de modèle peut permettre de révéler ce problème plus tôt dans le processus de développement.

3.5.2.4 Outils d'animation commerciaux de B

Dans les paragraphes précédentes, nous avons identifié des propriétés requises d'un outil d'animation dans le cadre du test des modèles abstraits. Voyons maintenant si les outils d'animation des ateliers commerciaux de développement B possèdent ces propriétés.

Les deux ateliers commerciaux de développement en B, Atelier B et B Toolkit, possèdent un outil d'animation. Nous avons brièvement discuté de ces outils dans le paragraphe 2.6.2. Ils sont basés sur des principes similaires : des prédicats et des expressions de la théorie des ensembles sont évalués en utilisant des règles de réécriture, souvent avec l'aide de l'utilisateur, et le résultat peut être montré sous une forme symbolique. L'indéterminisme est toujours levé par l'utilisateur à qui il est demandé d'entrer une valeur.

Un tel outil peut être utile pour aider le concepteur – et non pas le testeur – à avoir une vision plus approfondie de ce qui est spécifié par le modèle. Par exemple, avec l'outil d'animation de l'Atelier B, il est possible d'afficher la valeur des variables d'état en cours d'animation, ou encore la valeur des expressions ou des prédicats rentrés par l'utilisateur. L'outil d'animation étant connecté à l'outil de preuve, la terminaison des opérations ou la conservation de l'invariant peuvent être prouvées dans le contexte courant de l'animation. Cependant ces animateurs ne sont pas adaptés aux besoins de notre cadre unifié de test. En premier lieu, l'animation est limitée à un composant de type MACHINE et ne traverse pas les couches de raffinement. En second lieu, la correction de l'animation n'est pas garantie, car l'utilisateur est sollicité pour lever l'indéterminisme et il intervient également au cours de l'animation pour simplifier des prédicats et des expressions.

Comme nous l'avons mentionné dans le paragraphe 2.6.3, le problème de la correction d'une animation a été étudié dans [Breuer et Bowen 1994] pour la notation Z. Les auteurs argumentent que le résultat calculé au cours d'une animation est correct s'il est une approximation du résultat attendu exprimé par la sémantique de la spécification. Leur définition d'une approximation correcte prend en compte le fait que l'animation peut ne pas se terminer. La notion de sortie indéfinie est alors introduite par la notation ⊥.

Reprenons l'exemple de la séquence de test T ; *oo* ← Est_Un_Triangle (*1*, *2*, *2*), implémentée sur la MACHINE TRIANGLE, avec le résultat attendu T_0 ; *oo* := TRUE. Le résultat exprimé par la sémantique de l'opération Est_Un_Triangle, spécifiée dans la MACHINE TRIANGLE, est l'ensemble {TRUE, FALSE}. Les approximations correctes de ce résultat, dans l'ordre décroissant de précision, sont alors : {TRUE, FALSE}, {TRUE, FALSE}$_\perp$, {TRUE}$_\perp$ ou {FALSE}$_\perp$, et ⊥. Dans le premier cas, l'outil d'animation a pu calculer le résultat exact. Pour les trois cas suivants, l'outil a calculé un résultat partiel mais le calcul ne s'est pas terminé. Dans le dernier cas, l'outil n'a pu calculer de résultat. Il est à noter que ni {TRUE} ni {FALSE} ne constituent une approximation correcte de {TRUE, FALSE}.

Ces résultats correspondent aux caractéristiques de l'outil d'animation que nous avons identifiées pour les besoins du test. Dans notre exemple, si l'outil calcule un des résultats {TRUE, FALSE}, {TRUE, FALSE}$_\perp$ ou {FALSE}$_\perp$, l'expérience de test est concluante et conduira au rejet du résultat. Si l'outil calcule les résultats {TRUE}$_\perp$ ou ⊥, l'expérience de test n'est pas concluante.

3.5.3 Limites de la définition de la correction de test basé sur le raffinement

Notre cadre unifié définit l'oracle en termes d'une relation de raffinement, de façon à tirer profit du caractère formel de la méthode B : pour une séquence de test, l'acceptation des résultats fournis par le modèle obtenu à une étape de développement implique l'acceptation des résultats fournis par des raffinements corrects de ce modèle. Cependant, cette définition entraîne des limites. Nous discutons dans ce paragraphe du problème des entrées et des sorties interactives.

Les modèles B peuvent interagir avec l'extérieur à l'aide d'interfaces spécifiques. Ces interfaces sont fournies dans des bibliothèques de composants de type MACHINE qui ne sont pas raffinés, mais pour lesquels le code source dans un langage de programmation est déjà disponible. Indépendamment de la complexité du code associé, toutes ces interfaces ont une spécification abstraite similaire dans la notation B :

- Les opérations de lecture sont spécifiées par un choix indéterministe dans un ensemble de valeurs. Par exemple, l'opération de lecture d'un nombre entier appartenant à l'intervalle [*aa* .. *bb*], est définie dans le B-Book comme suit :

$xx \leftarrow$ INTERVAL_READ (aa, bb) ĩ PRE $aa \in$ NAT $\wedge$
$bb \in$ NAT $\wedge$ $aa \leq bb$
THEN $xx :\in aa .. bb$
END

Elle spécifie l'acquisition d'une valeur quelconque dans l'intervalle, indépendamment du dispositif d'entrée qui va être utilisé en pratique.

- Les opérations d'écriture sont spécifiées comme des substitutions pré-conditionnées qui se terminent quand le paramètre d'entrée a le type souhaité. Un exemple est :

INT_WRITE (xx) ĩ PRE $xx \in$ INT THEN
skip
END

La sémantique de ces interactions avec l'extérieur n'est pas couverte par les obligations de preuve de la méthode B. Prenons l'exemple de l'opération suivante :

incr ĩ BEGIN
VAR xx IN
$xx \leftarrow$ INTERVAL_READ (0, 9);
INT_WRITE ($xx + 1$)
END
END

La substitution obtenue par expansion des appels aux opérations, INTERVAL_READ et INT_WRITE étant remplacés par leur texte formel, est équivalente à skip : incr se termine toujours et une post-condition faible, TRUE, est établie (nous supposons que le composant ne contient pas de variables d'état). Des raffinements corrects de incr peuvent être :

STRING_WRITE ("Hello World") ou VAR xx IN
$xx \leftarrow$ INTERVAL_READ (0, 5);
INT_WRITE ($xx + 2$)
END

parmi beaucoup d'autres possibilités. Par conséquent, si le modèle sous test contient des opérations de lecture et d'écriture, on ne peut plus se référer aux obligations de preuve de raffinement pour conserver la correction.

Cette limite concerne des systèmes qui ont un comportement fortement interactif. La spécification et le test de tels systèmes communicants requièrent une notation et une sémantique adaptées. Typiquement, un système communicant et un scénario de test associé sont décrits par des processus connectés [Brinksma 1989; Gaudel et James 1998]. De même que la méthode B, le cadre unifié est adéquat pour des systèmes séquentiels dans lesquels nous pouvons distinguer trois phases d'acquisition de données, de traitement, et d'affichage des résultats. L'exemple du Triangle et l'étude de cas industrielle que nous présenterons dans les deux paragraphes suivants, fournissent des exemples de tels systèmes. Pour ce type de systèmes, la vérification

peut se concentrer sur la partie de traitement, des moniteurs et des bouchons simulant les phases d'entrée et de sortie.

3.6 Illustration du cadre théorique par l'exemple du Triangle

Dans le chapitre 2, nous avons introduit le développement B de l'exemple du Triangle pour mettre en évidence la possibilité d'introduire des fautes dans la spécification B, sans qu'elles soient révélées par les obligations de preuve. Nous utilisons ici cet exemple afin d'illustrer les deux aspects de notre cadre théorique, c'est-à-dire l'identification des différentes étapes du développement, et les séquences de test qui peuvent être implémentées sur les modèles obtenus à ces étapes.

3.6.1 Analyse préliminaire

Nous rappelons dans la Figure 3.10 l'architecture du développement de l'exemple du Triangle.

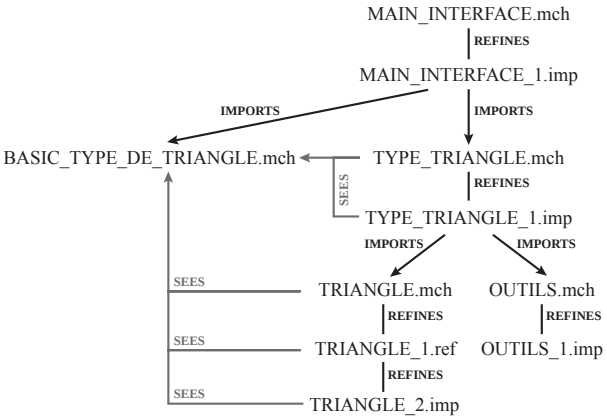


Figure 3.10. Architecture du développement de l'exemple du Triangle

Le composant racine de l'architecture est la MACHINE MAIN_INTERFACE qui offre une opération *main*. Nous donnons, dans la Figure 3.11, le texte de cette opération et son implémentation dans le composant MAIN_INTERFACE_1.imp. Dans le composant racine, l'opération *main* n'exprime qu'une vue boîte noire du programme (substitution skip). A ce niveau rien ne peut être observé ni commandé. L'implémentation de cette opération séquence les trois phases d'acquisition de données, de calcul et d'affichage des résultats : elle demande à l'utilisateur d'entrer trois nombres entiers ayant une valeur entre *0* et $\sqrt{\text{MAXINT}}$, calcule la validité et le type de triangle par un appel à l'opération Classe_Triangle, et affiche le résultat. Comme nous l'avons expliqué dans le paragraphe 3.5.3, le test de ce système peut se

concentrer sur la phase de calcul, les phases d'acquisition et d'affichage étant simulées par des bouchons. L'opération Classe_Triangle étant spécifiée dans le composant TYPE_TRIANGLE, nous allons donc tester le sous-système ayant ce composant pour racine.

main ĺ **skip**	**main** ĺ **BEGIN** **VAR** *mint*, *c1*, *c2*, *c3*, *tt* **IN** *mint* ← **SQRT** (**MAXINT**); **STRING_WRITE** ("Entrez un entier :"); *c1* ← **INTERVAL_READ** (*0*, *mint*); **STRING_WRITE** ("Entrez un entier :"); *c2* ← **INTERVAL_READ** (*0*, *mint*); **STRING_WRITE** ("Entrez un entier :"); *c3* ← **INTERVAL_READ** (*0*, *mint*); *tt* ← **Classe_Triangle** (*c1*, *c2*, *c3*); **STRING_WRITE** ("On a un triangle du type :"); **TYPE_DE_TRIANGLE_WRITE** (*tt*); **END** **END**

Figure 3.11. Spécification de l'opération main dans le composant MAIN_INTERFACE.mch et son implémentation dans le composant MAIN_INTERFACE_1.imp

Pour cela, un moniteur de test doit être construit sur le modèle de l'implémentation de *main*, les interfaces utilisées par ce moniteur (acquisition des entrées et observation des sorties) étant appropriées à l'environnement d'animation. Le moniteur importe le modèle déplié sous test de racine TYPE_TRIANGLE, pour pouvoir implémenter les séquences de test sur ce modèle. Il doit également importer un bouchon simulant le composant vu BASIC_TYPE_DE_TRIANGLE, qui définit : 1) l'ensemble TYPE_DE_TRIANGLE = {ISOCELE, EQUILATERAL, SCALENE, RECTANGLE, INVALIDE}, et 2) les opérations de lecture et d'écriture concernant les éléments de cet ensemble (par exemple, l'opération TYPE_DE_TRIANGLE_WRITE).

Nous allons maintenant examiner le sous-ensemble du développement à partir du composant TYPE_TRIANGLE, pour identifier les étapes auxquelles des modèles testables sont obtenus.

3.6.2 Identification des étapes de développement

En suivant les conditions architecturales définies dans les paragraphes 3.3.2 et 3.3.3, les différentes étapes de ce développement peuvent être identifiées de manière systématique. Dans la Figure 3.12, nous montrons la hiérarchie des modèles correspondants. Chaque modèle est constitué d'un ensemble de composants dont la mise à plat résulte en une machine abstraite. Ainsi, le modèle de plus haut niveau est la MACHINE TYPE_TRIANGLE. Le Modèle 2 contient en plus le raffinement de cette

MACHINE, c'est-à-dire l'IMPLEMENTATION TYPE_TRIANGLE_1, ainsi que les composants importés TRIANGLE et OUTILS. Selon les conditions architecturales, le Modèle 2 est un raffinement du Modèle 1, et est raffiné par l'implémentation finale. Dans cet exemple, il n'y a pas de lien SEES interne au sous-système considéré, et donc pas d'ordre imposé pour le raffinement des différentes branches du développement. Ainsi, n'importe lequel des Modèles 3 et 4 peut être pris comme successeur du Modèle 2, selon que l'on descende d'un niveau de raffinement à partir du composant TRIANGLE ou du composant OUTILS. Notons que la Figure 3.12 résulte de l'analyse a posteriori d'une architecture complète, allant jusqu'à l'implémentation finale. Une analyse similaire pourrait être conduite en cours de développement, donnant par exemple l'identification des Modèles 1 à 3.

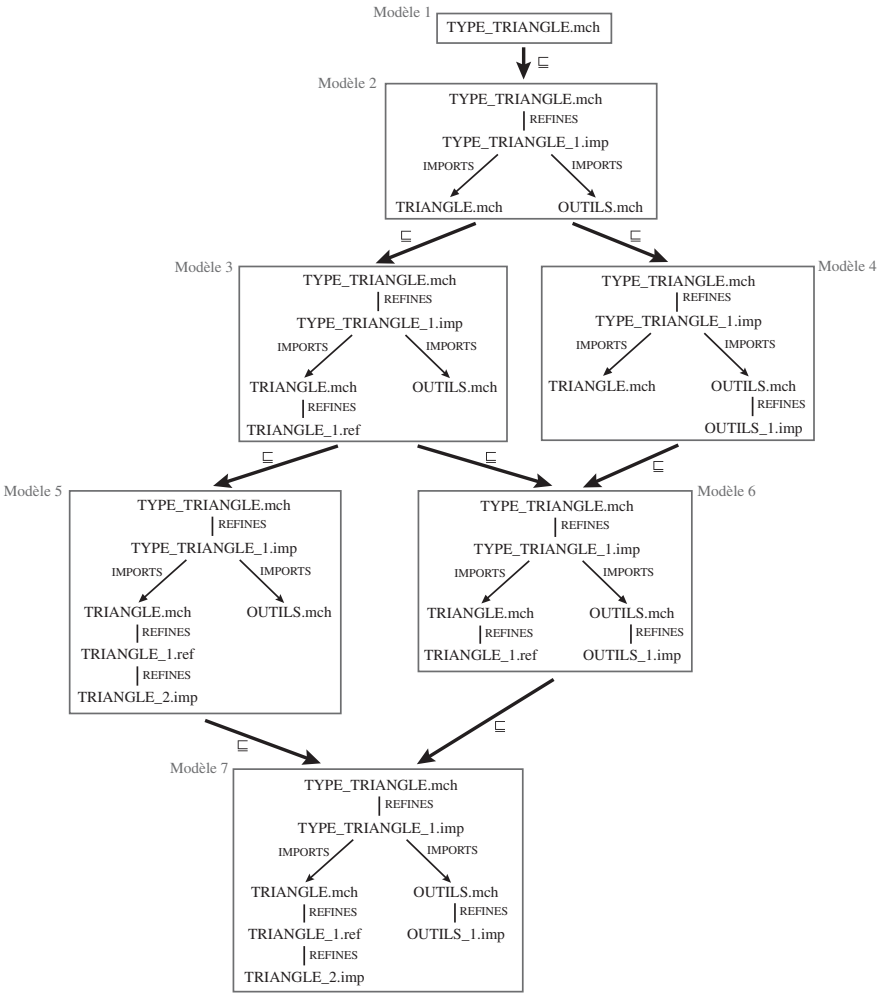


Figure 3.12. Modèles obtenus aux étapes de développement du sous-système de racine TYPE_TRIANGLE

Une fois qu'une hiérarchie de modèles est obtenue, se pose le choix du modèle à tester. Ce choix peut être guidé par une analyse de traçabilité vis-à-vis du cahier des charges, pour déterminer l'étape à laquelle tous les besoins de l'utilisateur sont censé avoir été pris en compte. Pour l'exemple du Triangle, une telle analyse de traçabilité a été présentée dans le chapitre 2 de ce mémoire (dans le paragraphe 2.5.3). Elle montre que la spécification complète du problème n'est obtenue qu'à la fin du développement, c'est-à-dire dans le Modèle 7. Ainsi, nous savons déjà que tous les modèles sont incomplets vis-à-vis des besoins du cahier des charges. Dans ce cas, la vérification externe est réalisée par test du code final, qui constitue une version compilée du modèle le plus concret.

Nous allons pourtant considérer, dans le paragraphe suivant, le test de chacun des modèles B de la hiérarchie. Ceci nous permettra d'illustrer comment se traduit l'imprécision de ces modèles en termes de réponses à des séquences de test. Nous verrons également comment la relation de raffinement est exploitée pour ne rejouer sur un modèle, qu'un sous-ensemble des séquences de test appliquées à ses abstractions.

3.6.3 Exemples de séquences de test et résultats obtenus

A titre illustratif, nous avons choisi six séquences de test. Conformément à notre cadre unifié, ces séquences peuvent être implémentées sur n'importe lequel des modèles du développement. Dans le Tableau 3.1, nous montrons ces séquences de test ainsi que le résultat attendu pour chaque séquence. Aucun des composants n'ayant de variables d'état, une séquence de test n'est constituée que d'un appel à l'opération Classe_Triangle. Nous allons voir par la suite quel est le résultat obtenu à partir de l'implémentation de ces séquences sur chaque modèle, et si ce résultat correspond à celui exprimé par T_0. Pour calculer le résultat de l'implémentation de chaque séquence de test, nous nous référons à l'analyse de la spécification de l'opération Classe_Triangle et ses raffinements, présentée dans le paragraphe 2.5.3.

Tableau 3.1. Séquences de test et résultats attendus pour l'exemple du Triangle

Séquence de test	Résultat attendu
T1 ; *res* ← Classe_Triangle (*0*, *0*, *0*)	$T1_0$; *res* := INVALIDE
T2 ; *res* ← Classe_Triangle (*2*, *3*, *4*)	$T2_0$; *res* := SCALENE
T3 ; *res* ← Classe_Triangle (*3*, *4*, *5*)	$T3_0$; *res* := RECTANGLE
T4 ; *res* ← Classe_Triangle (*2*, *2*, *3*)	$T4_0$; *res* := ISOCELE
T5 ; *res* ← Classe_Triangle (*1*, *2*, *3*)	$T5_0$; *res* := INVALIDE
T6 ; *res* ← Classe_Triangle (*1*, *1*, *1*)	$T6_0$; *res* := EQUILATERAL

La spécification de Classe_Triangle dans le Modèle 1 est l'affectation indéterministe d'un élément de l'ensemble TYPE_DE_TRIANGLE au paramètre de sortie

(voir la Figure 2.7.a). Le résultat de l'implémentation de toutes les séquences de test sur ce modèle est le suivant :

$$\text{pour tout } i \in \{1..6\},\quad Ti_{Modèle\ 1} = res :\in \text{TYPE\_DE\_TRIANGLE}$$

D'après notre définition de l'oracle de test, aucun de ces résultats ne peut être accepté car ils ne raffinent pas les résultats attendus exprimés par les Ti_0. En effet, nous avons la relation de raffinement suivante (les modèles n'ayant pas de variables d'état la substitution $@Ti_{Modèle\ 1}$ est équivalente à $Ti_{Modèle\ 1}$) :

$$\text{pour tout } i \in \{1..6\},\ Ti_{Modèle\ 1} \quad \Diamond \quad Ti_0$$

Cela veut dire que les résultats calculés par le Modèle 1 sont plus indéterministes que ceux exprimés par les Ti_0. Aucun incohérence n'a été observée, mais le modèle n'est pas suffisamment précis vis-à-vis des besoins fonctionnels et ses raffinements doivent être considérés.

Pour le Modèle 2, la seule séquence de test qui calcule le résultat correct est la séquence *T1*. En effet, $T1_{Modèle\ 2} = res$:= INVALIDE. Pour les autres séquences, le résultat obtenu est toujours indéterministe. Nous détaillons, par exemple, le résultat obtenu à partir de l'implémentation de la séquence *T4* sur ce modèle, dans laquelle Classe_Triangle est appelé avec les valeurs *2*, *2* et *3* (cas isocèle). Ces valeurs sont d'abord ordonnées dans l'ordre croissant par un appel à l'opération Ordonne_3_NAT spécifiée dans le composant OUTILS (voir le Figure 2.7.b). La spécification de Ordonne_3_NAT est indéterministe (voir la Figure 2.7.c), et pour les valeurs *2*, *2* et *3* calcule les résultats suivants : a) *2*, *2*, *2* ; b) *3*, *3*, *3* ; c) *2*, *3*, *3* et d) *2*, *2*, *3*. Ces valeurs correspondraient à un triangle équilatéral (pour les résultats a et b) ou isocèle (pour les résultats c et d). Ensuite, l'opération Est_Un_Triangle vérifie si les valeurs renvoyées par Ordonne_3_NAT peuvent constituer un triangle valide. La spécification de cette opération étant également indéterministe (voir la Figure 2.7.d), elle renvoie à la fois le résultat TRUE et FALSE. Enfin, l'opération Quel_Triangle calcule le type de triangle. Cette opération renvoie de manière indéterministe un élément de TYPE_DE_TRIANGLE (voir la Figure 2.7.d). Le résultat de l'implémentation de la séquence *T4* sur le Modèle 2 est finalement le suivant :

$$T4_{Modèle\ 2} = res :\in \text{TYPE\_DE\_TRIANGLE}$$

Le résultat de l'implémentation de chaque séquence de test sur chaque modèle peut être calculé de la même manière. Nous récapitulons ces résultats dans le Tableau 3.2. Le résultat de l'implémentation de la séquence de test *T1* sur le Modèle 2 étant accepté, nous avons la garantie que tout raffinement de ce modèle fournira également un résultat correct en réponse à *T1*. Cette séquence de test n'est alors plus considérée. De même, la séquence *T5* n'a pas besoin d'être rejouée sur le Modèle 6, etc. Le Modèle 7, qui correspond à l'implémentation finale, calcule le résultat correct pour toutes les séquences de test restantes.

Tableau 3.2. Les résultats de l'implémentation des séquences de test sur les modèles du développement.

Modèle	Résultat de l'implémentation de séquence de test
Modèle 3	$T2_{Modèle\ 3} = res :\in$ TYPE_DE_TRIANGLE $T3_{Modèle\ 3} = res :\in$ TYPE_DE_TRIANGLE $T4_{Modèle\ 3} = res :\in$ {INVALIDE, ISOCELE, EQUILATERAL} $T5_{Modèle\ 3} = res :\in$ {INVALIDE, ISOCELE, EQUILATERAL} $T6_{Modèle\ 3} = res :\in$ {INVALIDE, EQUILATERAL}
Modèle 4	$T2_{Modèle\ 4} = res :\in$ TYPE_DE_TRIANGLE $T3_{Modèle\ 4} = res :\in$ TYPE_DE_TRIANGLE $T4_{Modèle\ 4} = res :\in$ TYPE_DE_TRIANGLE $T5_{Modèle\ 4} = res :=$ INVALIDE accepté par l'oracle $T6_{Modèle\ 4} = res :\in$ TYPE_DE_TRIANGLE
Modèle 5	$T2_{Modèle\ 5} = res :\in$ {SCALENE, EQUILATERAL, ISOCELE, INVALIDE} $T3_{Modèle\ 5} = res :\in$ {SCALENE, EQUILATERAL, ISOCELE, INVALIDE} $T4_{Modèle\ 5} = res :\in$ {EQUILATERAL, ISOCELE} $T5_{Modèle\ 5} = res :\in$ {EQUILATERAL, ISOCELE, INVALIDE} $T6_{Modèle\ 5} = res :\in$ EQUILATERAL accepté par l'oracle
Modèle 6	$T2_{Modèle\ 6} = res :\in$ {SCALENE, RECTANGLE, INVALIDE} $T3_{Modèle\ 6} = res :\in$ {SCALENE, SCALENE, INVALIDE} $T4_{Modèle\ 6} = res :\in$ {SCALENE, INVALIDE} $T6_{Modèle\ 6} = res :\in$ {EQUILATERAL, INVALIDE}

Nous avons vu à l'aide de cet exemple, comment nous pouvons raisonner sur le comportement observable des modèles B dépliés en réponse à des séquences de test. Ce raisonnement permet d'avoir une vision plus claire de ce que l'on a spécifié à une étape donnée. Par exemple, contrairement à ce que l'on pourrait croire, le comportement spécifié dans le Modèle 2 n'est pas beaucoup plus précis que celui spécifié dans le Modèle 1 qui ne fait que du typage : le seul cas qui a été précisé est le retour de la valeur INVALIDE pour les entrées *0*, *0*, *0*. Même aux étapes tardives correspondant aux Modèles 5 et 6, la plupart des cas de type de triangle ne peuvent être déterminés.

Analyser le comportement d'un modèle en termes d'entrées et de sorties peut ainsi permettre non seulement de révéler des fautes (incohérence entre résultats observés et attendus), mais aussi d'identifier des imprécisions de ce modèle, laissant la possibilité d'introduire des fautes dans les étapes ultérieures du développement. Dans ce cas, les résultats obtenus indiquent quelles séquences devront être rejouées, afin de valider les choix de raffinement qui suppriment ces imprécisions.

3.7 Etude de cas industrielle

Afin de mettre en pratique les concepts liés à notre cadre théorique de test, nous avons également étudié un cas réel, issu de l'industrie ferroviaire. Le logiciel étudié est le logiciel MPL75 développé par Alstom Transport SA. Il constitue le logiciel ATP[1] embarqué assurant le contrôle en sécurité des trains sur les lignes A et B du métro de Lyon. C'est un programme séquentiel, déterministe et non-distribué, activé cycliquement par un noyau temps-réel. Le programme lit ses entrées et écrit ses sorties dans les registres des constituants matériels qui réalisent l'interfaçage avec le train.

Le développement B de ce logiciel est constitué de 51 composants et traverse 13 niveaux de raffinement et de décomposition. Il nous a été fourni avec le cahier des charges du logiciel. Ce document décrit les fonctionnalités du logiciel ainsi que les contraintes qu'il doit respecter, ses modes de fonctionnement et ses interfaces avec le reste du système. Le cahier des charges contient également une analyse de type SADT qui présente un premier découpage fonctionnel du logiciel en sous-systèmes.

L'analyse de l'architecture B nous a permis d'identifier des étapes de développement en suivant les conditions architecturales que nous avons présentées dans les paragraphes 3.3.2 et 3.3.3. En nous basant sur le cahier des charges, nous avons ensuite cherché à distinguer l'étape correspondant à la fin de la phase de conception préliminaire en B. Cette étape nous fournit donc le plus petit modèle censé exprimer tous les besoins fonctionnels du cahier des charges, les raffinements ultérieurs n'introduisant que des détails de programmation. C'est ce modèle qui, à notre avis, doit constituer la cible du test. Enfin, une analyse en termes d'interface nous a permis d'identifier la forme générale des séquences de test implémentables sur ce modèle. Nous présentons brièvement les résultats de ces analyses ci-dessous.

3.7.1 Etapes de développement

Mentionnons tout d'abord que cette application vérifie la condition de [Rouzaud 1999] (voir le paragraphe 3.3.3.1) renforcée avec nos conditions C1 et C2 (voir le paragraphe 3.3.3.2), restreignant l'utilisation du lien SEES. Nous avons également pu constater que les sous-systèmes issus de l'analyse SADT, que l'on retrouve dans l'architecture B, satisfont les conditions pour pouvoir être testés indépendamment (voir le paragraphe 3.3.3.3) : ils n'introduisent pas de dépendances croisées par liens SEES, et les composants extérieurs à un sous-système, vus par celui-ci, constituent la racine de sous-arbres mutuellement disjoints. Les conditions architecturales que nous avons définies ne semblent donc pas incompatibles avec la pratique industrielle. Sur

[1] ATP (*Atutomatic Train Protection*) : système surveillant la correspondance entre la dynamique du train (vitesse, position, etc.) et la signalisation reçue du sol, et pouvant déclencher le freinage d'urgence en situation dangereuse [Dehbonei et Mejia 1995]

cet exemple, on peut définir des étapes de développement aussi bien pour l'application complète que pour ses sous-systèmes.

Le développement B du logiciel commence par une machine abstraite de haut niveau comportant une opération *main*. La spécification de l'opération *main* à cette étape est très simple et donne une description très abstraite du comportement du logiciel. Tout ce qui est spécifié est que les variables peuvent être modifiées et que leurs nouvelles valeurs conservent l'invariant. Cet invariant décrit quelques contraintes globales du logiciel, d'autres propriétés de sécurité du logiciel étant spécifiées au fur et à mesure des raffinements dans des machines abstraites de plus bas niveau dans l'architecture.

Dans le composant racine, l'opération *main* n'exprime qu'une vue boîte noire du comportement du logiciel. Pour avoir les phases d'acquisition de données, de traitement et de sortie, nous devons considérer l'implémentation de cette opération. Le composant racine est implémenté en important 1) des composants MACHINEs spécifiant des interfaces d'entrée et de sortie, 2) des composants MACHINEs spécifiant les données statiques (constantes) de l'application, et 3) des composants MACHINEs spécifiant chacun des sous-systèmes issus de la décomposition SADT décrite dans le cahier des charges. A cette étape, nous n'avons pas encore un modèle de toutes les fonctionnalités attendues. Pour deux sous-systèmes, la phase de conception préliminaire en B traverse plusieurs niveaux de raffinement et de décomposition en couches. A titre d'exemple, la Figure 3.13 donne l'architecture symbolique d'un des sous-systèmes, chargé de contrôler le mouvement du train. Pour ce sous-système, l'ensemble des composants supposé inclure toutes les exigences du cahier des charges, d'après notre analyse, apparaît en grisé sur la figure.

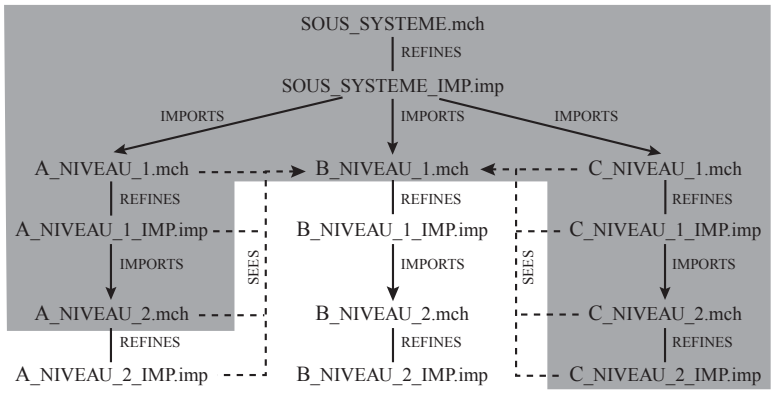


Figure 3.13. Architecture symbolique d'un des sous-systèmes de l'étude de cas.

3.7.2 Commandabilité et observabilité des modèles dépliés

Comme dans le cas du Triangle, l'implémentation du composant racine peut être vue comme un moniteur qui connecte l'application à ses interfaces d'entrée/sortie (le code

des interfaces étant obtenu en dehors de la méthode B), et séquence les phases d'acquisition, de traitement et de sortie. Le test de ces traitements peut donc être piloté par un moniteur similaire à l'implémentation du composant racine, où les interfaces sont simulées par des composants permettant de commander les entrées de test, et d'observer les sorties. Il est à noter que l'utilisation du code réel des interfaces n'a de sens que pour une étape de test ultérieure, en intégration avec le matériel.

Dans cette application, chaque sous-système offre une seule opération, correspondant à une fonction de haut niveau identifiée par l'analyse SADT. L'ordre d'appel de ces opérations dans un cycle est immuable. Une séquence de test du système complet est donc constituée de l'initialisation, suivie d'un nombre fini de cycles : acquisition, appel aux opérations dans l'ordre spécifié, observation des sorties.

De façon générale, les entrées des opérations sont fournies par des liens SEES plutôt que par passage de paramètres. Dans l'exemple de la Figure 3.13, l'opération a un seul paramètre d'entrée, correspondant à une valeur calculée au cycle précédent par un autre sous-système. Mais il existe des liens SEES (non visualisés sur la figure) de tous les composants du sous-système vers des interfaces d'acquisition et vers des sous-systèmes fonctionnels dont l'opération précède celle du sous-système dans le cycle. Comme nous l'avons précisé, les composants vus sont racines de sous-arbres mutuellement disjoints. Les liens SEES sont introduits dans le composant racine du sous-système, étant ainsi dans la portée du lien IMPORTS de la racine de l'application. Ces liens SEES sont conservés lors du dépliage de modèle. Ils permettent de consulter les valeurs d'entrée du sous-système, qui sont toujours stockées dans des variables concrètes du composant vu. De la même manière, les sorties calculées par ce sous-système sont écrites dans ses variables concrètes. L'opération ne retourne rien, et l'interface de sortie, ainsi que les autres sous-systèmes utilisant les valeurs calculées, doivent voir (lien SEES) le composant concerné.

Que le test cible l'application complète (hors interfaces avec le matériel), ou un sous-système de celle-ci, le moniteur de test doit donc commander les entrées de test via des bouchons simulant chaque composant vu.

Les sorties calculées par le modèle cible (application complète, ou un sous-système) sont stockées dans ses variables concrètes. Comme nous l'avons mentionné dans le paragraphe 3.5.1, ceci ne constitue pas un problème vis-à-vis de notre cadre théorique. Ces variables concrètes peuvent être consultées par le moniteur de test. Elles sont garanties exister dans tous les raffinements du modèle sous test, y compris l'implémentation finale, et les obligations de preuve liées aux invariants de liaison permettent d'assurer que les post-conditions établies pour ces variables (notamment, la conformité par rapport aux résultats attendus spécifiés dans T_0) sont préservées lors des raffinements.

3.8 Quelques règles méthodologiques

La définition du cadre de test nous a amené à considérer le processus de développement en B du point de vue de la testabilité des modèles formels. Dans ce paragraphe, nous proposons quelques règles méthodologiques pour le développement des logiciels en B, qui prennent en compte la possibilité d'effectuer des tests.

Mentionnons tout d'abord un problème général à la méthode B : une utilisation maîtrisée des liens SEES est nécessaire pour éviter des architectures dans lesquelles tous les composants sont prouvés corrects, mais l'implémentation finale viole la spécification de départ. Comme nous l'avons brièvement rappelé dans le paragraphe 3.3.3.1, des conditions architecturales ont été proposées par [Rouzaud 1999] pour répondre à ce problème. La satisfaction de ces conditions est suffisante pour garantir la cohérence d'un développement prouvé, et nous ne pouvons que recommander leur prise en compte dans une méthodologie de conception en B. En fait, nous sommes même amenés à préconiser des restrictions plus fortes sur l'utilisation des liens SEES.

La technique de test que nous avons proposée s'inscrit dans un objectif de vérification externe. Ces tests ont pour but de révéler des fautes dans la spécification B, vis-à-vis des besoins fonctionnels décrits dans le cahier des charges. Ils sont d'autant plus efficaces qu'ils sont effectués tôt dans le développement, avant que le code exécutable ne soit généré. Cependant, ceci suppose qu'on puisse identifier, dans un développement incomplet, des modèles formels dont le comportement soit représentatif – selon la relation de raffinement – du comportement de l'implémentation finale. Or, l'identification de tels modèles ne peut être effectuée qu'à la fin, avec la connaissance de tous les liens SEES : le contexte dans lequel les opérations sont testées doit tenir compte du caractère modifiable ou consultable des variables dans l'architecture complète. La discussion de ce problème (voir le paragraphe 3.3.3.2) nous a conduit à n'autoriser l'introduction de nouveaux liens SEES que dans les deux cas suivants : 1) pour relier deux composants MACHINE importés dans une même IMPLEMENTATION ; 2) pour relier un composant quelconque à une machine qui ne sera pas raffinée (par exemple, MACHINE qui spécifie l'interface entre le programme B et son environnement). Il devient alors possible d'analyser une architecture incomplète en termes d'étapes de développement, auxquelles des modèles représentatifs de l'implémentation future sont obtenus.

Pour que le test d'un modèle, obtenu à une étape intermédiaire de développement, puisse être concluant, ce modèle doit exprimer une portion significative des besoins du cahier des charges. En B, aucun mécanisme explicite ne permet de séparer la phase de conception préliminaire, qui consiste à transcrire les besoins de l'utilisateur dans un modèle formel, de la phase de conception détaillée, dans laquelle le modèle formel est raffiné pour obtenir une version concrète proche d'un programme. Comme nous l'avons vu dans l'exemple du Triangle, les besoins peuvent être rentrés tout au long du processus de développement, jusqu'à la version la plus concrète. Un exemple plus significatif d'une telle approche est le développement en B du programme de contrôle/commande d'une chaudière, présenté dans le B-Book [Abrial 1996]. Cependant, une approche différente a été adoptée dans l'industrie ferroviaire

française, qui consiste à exprimer tous les besoins fonctionnels dans une phase intermédiaire du développement. L'étude de cas présentée dans le paragraphe précédent en fournit un exemple. Dans la mise en œuvre d'un développement en B, effectuer cette distinction entre conception préliminaire et conception détaillée nous paraît souhaitable. Dans ce cas, le modèle correspondant à la fin de la conception préliminaire constitue une cible privilégiée de la vérification externe, qu'elle s'effectue par test ou par d'autres techniques (par exemple, model-checking).

Le choix du modèle censé exprimer les besoins de l'utilisateur est guidé par une analyse de traçabilité du cahier des charges. Cependant, dans cette analyse, les composants ne peuvent pas être considérés séparément. Le modèle choisi doit pouvoir être associé à une étape de développement. Ce modèle doit alors vérifier les conditions architecturales définies dans les paragraphes 3.3.2 et 3.3.3. D'après ces conditions, l'existence de liens SEES impose un ordre pour le raffinement des différentes branches du développement. Prenons l'exemple de la Figure 3.6.b, avec T_1 le sous-arbre de racine A et de feuilles B_I_1, B_2. Comme nous l'avons expliqué dans le paragraphe 3.3.3.2, $\Phi(T_1)$ n'est pas raffiné par l'implémentation finale et il ne peut alors pas être attribué à une étape de développement. Dans cette architecture, si le composant B_1 n'est pas suffisamment précis par rapport aux besoins, son implémentation B_I_1 doit être considérée. Nous serons ainsi obligé d'inclure dans le modèle tous les raffinements de B_2, jusqu'au composant B_4. Pour un autre exemple d'ordre imposé par le lien SEES, nous utilisons l'architecture de la Figure 3.3. Dans cette architecture, le sous-arbre de feuilles M_1, M_R_2 n'appartient pas au domaine d'entrée valide de la fonction de dépliage. Si le composant M_2 n'est pas suffisamment précis vis-à-vis des besoins et si son raffinement doit être inclus, le modèle doit également contenir l'implémentation du composant M_1. Notons que, dans cet exemple, l'ordre est imposé à cause d'un problème syntaxique concernant les règles de visibilité du lien SEES. On pourrait envisager de relâcher cet ordre si le composant vu ne contenait pas de variables d'état abstraites. Mais il faudrait alors apporter la démonstration que la machine M_1, origine du liens SEES, est plongée dans un contexte représentatif de celui de l'implémentation finale : ce sera le cas si les variables de M_R_2 non modifiées par M_1 ne sont pas non plus modifiées par les raffinements et décompositions de M_1.

Dans le cas du test de sous-systèmes, l'existence de références croisées par liens SEES ne permet pas que ces sous-systèmes soient testés indépendamment. Si le développement de l'application globale peut être décomposé en développement de sous-systèmes, nous recommandons de prévoir cette décomposition dès les premières étapes de développement. Par exemple, dans l'étude de cas industrielle mentionnée dans le paragraphe 3.7, le découpage fonctionnel de l'application en sous-systèmes a été effectué dans le cahier des charges. Dans le développement B, ces sous-systèmes sont spécifiés dès l'implémentation du composant racine. Les dépendances par liens SEES entre sous-systèmes sont toutes introduites dans la portée du IMPORTS effectué par cette implémentation. Ces dépendances sont unidirectionnelles, et tous les composants vus par un sous-système le sont dès le composant racine de ce sous-système.

Afin de pouvoir tester les modèles obtenus aux étapes intermédiaires du développement, nous devons pouvoir commander ces modèles et observer les résultats fournis par ceux-ci. Comme nous l'avons expliqué dans le paragraphe 3.5.3, notre cadre de test est adéquat pour les systèmes dans lesquels trois phases d'acquisition, de traitement et de sorties peuvent être identifiés. Pour de tels systèmes, le test se concentre sur la partie de traitement en fournissant des bouchons pour les parties d'acquisition et de sortie. Pour cela, les interfaces d'entrée et de sortie doivent être spécifiées dès les premières phases de développement, typiquement au niveau de l'implémentation du composant racine. De même, dans le cas du test des sous-systèmes, nous recommandons de spécifier les entrées et les sorties dans le composant racine de chaque sous-système. Par exemple, dans l'étude de cas industrielle, les sorties calculées par un sous-système sont stockées dans des variables concrètes spécifiées dès le composant racine. Elles sont donc observables à n'importe quelle étape de développement de ce sous-système. De plus, ces variables peuvent servir d'entrée à d'autres sous-systèmes, qui les consultent via des liens SEES, eux-mêmes introduits dès les racines de ces sous-systèmes.

3.9 Conclusion

Nous avons présenté dans ce chapitre le cadre théorique que nous avons proposé pour le test des modèles B.

Nous avons défini des conditions architecturales qui permettent d'identifier les différents modèles construits au cours d'un développement en B et d'ordonner ces modèles selon la relation de raffinement. Le processus de développement en B est ainsi explicitement représenté comme une succession d'étapes où des modèles de plus en plus concret de l'application sont construits. Ce sont ces modèles qui pourront faire l'objet de test.

Nous avons ensuite formalisé les notions de séquence de test et d'oracle pour ces modèles. Le cadre théorique de test que nous avons défini est un cadre unifié : il est indépendant du fait que les séquences de test soient exécutées sur les modèles formels ou sur le code source, vu comme une version compilée du modèle le plus concret. Dans le premier cas, les résultats sont obtenus par l'animation de modèle, et dans le second par l'exécution de code. L'oracle de test est défini en relation avec la notion de raffinement des machines abstraites en B, ce qui permet de tirer profit du caractère formel de la méthode B : pour une séquence de test, l'acceptation des résultats obtenus à partir d'un modèle implique l'acceptation des résultats obtenus à partir des raffinements prouvés de ce modèle.

Les modèles B que nous considérons sont en fait des machines abstraites « aplaties ». Nous avons défini un algorithme qui, à partir d'un ensemble de composants représentant une partie de l'architecture du développement, construit le texte formel déplié correspondant. Cette notion de mise à plat de modèles B, introduite pour les besoins du test, peut s'avérer utile dans un cadre plus général. Elle

permet de raisonner formellement sur le comportement observable spécifié par un ensemble de composants. Comme nous l'avons illustré sur l'exemple du Triangle, ceci permet par exemple de comparer, pour une séquence de test, les résultats obtenus à partir de modèles de différents niveaux d'abstraction. Ainsi, nous pouvons analyser les nouvelles fonctionnalités qui sont rentrées à chaque étape du développement. Le dépliage des liens est également intéressant du point de vue de la vérification formelle (preuve, model-checking) de propriétés transversales impliquant plusieurs composants. De plus, lorsque le modèle aplati correspond à la dernière étape de développement, on peut envisager qu'il serve de base à la génération de code source optimisé.

Enfin, la définition du cadre unifié de test nous a permis de proposer quelques règles méthodologiques, qui prennent en compte la testabilité des modèles construits au cours d'un développement en B.

Chapitre 4

Couverture structurelle de modèles B

4.1 Introduction

Après avoir établi un cadre formel pour le test de modèles B dépliés, nous nous intéressons dans ce chapitre au problème de la sélection de jeux de test.

Un test exhaustif de l'ensemble des séquences de test implémentables sur un modèle B n'est évidemment pas faisable. Des critères de sélection doivent alors être définis, qui spécifient un ensemble d'éléments à couvrir pendant le test. De même que pour le test de programmes, ces critères de test peuvent être définis à partir d'une analyse fonctionnelle des besoins exprimés dans le cahier des charges, ou d'une analyse structurelle des modèles B, ou des deux. Une approche typique peut être de compléter des jeux de test fonctionnels afin d'arriver à une couverture structurelle cible des modèles B. Cependant, la notion de couverture structurelle de modèles B nécessite quelques développements théoriques : dans le cadre unifié de test que nous avons défini, nous devons avoir des critères de couverture applicables à la fois aux modèles abstraits et concrets. Notre contribution au problème de la sélection de jeux de test porte sur ce point.

Rappelons que l'analyse structurelle de modèles B en vue du test comporte deux aspects (voir chapitre 2) : 1) analyse des opérations du modèle, pour extraire un ensemble de cas à couvrir pour chaque opération ; 2) analyse des séquencements possibles des appels d'opération, pour prendre en compte l'état interne encapsulé. Nous nous sommes concentrés sur le premier aspect : étant donné un modèle d'un niveau d'abstraction quelconque, il s'agit donc de définir des critères de couverture de ses opérations.

Dans la littérature, les critères existants diffèrent selon que l'on cherche à couvrir un programme (analyse du graphe de contrôle) ou une spécification orientée modèle (analyse du prédicat avant-après). Une unification de ces deux catégories d'approche est présentée dans le paragraphe 4.2. Cette unification est possible pour les modèles B car les concepts de spécification, raffinement et implémentation sont exprimés à l'aide d'une notation unique. Dans le paragraphe 4.3, nous proposons un ensemble de critères de couverture, partiellement ordonnés selon leur sévérité. L'ensemble des travaux présentés dans ces paragraphes a fait l'objet d'une publication [Behnia et Waeselynck 1999].

Enfin, nous terminons par une discussion sur les limites d'application des différents critères de couverture proposés (paragraphe 4.4), en prenant notamment l'exemple de l'étude de cas industrielle mentionnée dans le chapitre 3.

4.2 Unification des approches structurelles

Le cadre unifié de test s'applique aux modèles B dépliés, obtenus aux différentes étapes de développement. Comme nous l'avons mentionné dans le chapitre précédent, le modèle le plus abstrait, c'est-à-dire le modèle obtenu avant tout raffinement, n'est pas significatif vis-à-vis des propriétés que l'on cherche à vérifier. Ainsi, le modèle qui fera l'objet du test, obtenu à une étape ultérieure du développement, peut contenir des constructions abstraites de la notation B (substitution indéterministe, parallèle, etc.), des constructions programmables (condition IF, boucle WHILE, etc.), ou un mélange des deux.

Les approches de couverture de spécifications orientées modèles sont basées sur la structure des prédicats avant-après. D'un point de vue théorique, ces approches peuvent être appliquées aux modèles B quel que soit leur niveau d'abstraction. Nous discutons d'une telle application dans le paragraphe 4.2.1. Les approches structurelles définies pour les programmes procéduraux, et basées sur la sélection de chemins dans le graphe de contrôle, peuvent être appliquées aux modèles B les plus concrets qui contiennent uniquement des constructions programmables (paragraphe 4.2.2).

Nous proposons ensuite une extension de la notion de graphe de contrôle (paragraphe 4.2.3). Cette extension peut être appliquée aux constructions abstraites du langage des substitutions généralisées. Elle nous servira de base pour l'analyse structurelle des opérations d'un modèle obtenu à une étape quelconque du développement. Nous montrons dans le paragraphe 4.2.4 comment la couverture de "chemins" dans ce graphe de contrôle peut être liée à la couverture de la structure propositionnelle du prédicat avant/après de l'opération considérée.

4.2.1 Opérations B en tant que prédicats avant-après

Dans le paragraphe 2.4, nous avons présenté des travaux relatifs à l'analyse structurelle des spécifications orientées modèle. Les approches proposées par ces travaux sont basées sur la structure des prédicats avant-après : une analyse de partition des prédicats avant-après décompose le domaine d'entrée de chaque opération en sous-domaines ; les critères de test exigent alors qu'au moins une entrée de test soit générée pour chaque sous-domaine. Bien que ces approches soient proposées pour la couverture des spécifications écrites en VDM [Dick et Faivre 1993] et en Z [Hörcher et Peleska 1995; Hierons 1997], elles peuvent être appliquées à d'autres spécifications orientées modèle parmi lesquelles les spécifications B. Un exemple de ce type de traitement dans le cadre des spécifications B est présenté dans [Van Aertryck et al. 1997b], que nous avons mentionné dans le chapitre 2.

Il existe une transformation systématique des substitutions généralisées en prédicats avant-après, définie par le prédicat prd (voir le paragraphe 1.3.3). En utilisant cette transformation, nous pouvons appliquer l'analyse de partition sur les prédicats avant-après ainsi obtenus.

Sans perdre en généralité, nous pouvons supposer que l'opération analysée a la forme suivante :

PRE *P* THEN *S* END

Similairement à l'approche de [Dick et Faivre 1993], nous supposons que les entrées de test générées pour chaque opération devront satisfaire l'invariant et la pré-condition de l'opération. De plus, pour que le prédicat avant-après ait un sens, nous devons également considérer les informations concernant le contexte de la machine abstraite. Le contexte de la machine abstraite, utilisé également dans la définition des obligations de preuve (voir le paragraphe 1.6), contient la définition et les propriétés des ensembles et des constantes de la machine.

Les prédicats avant-après portent à la fois sur l'état avant, l'état après, les paramètres d'entrée et les paramètres de sortie de l'opération. Comme nous l'avons vu pour [Dick et Faivre 1993; Hörcher et Peleska 1995; Hierons 1997], une quantification existentielle sur l'état après et les paramètres de sortie doit être utilisée afin d'obtenir une partition du domaine d'entrée uniquement. Pour les substitutions généralisées, cette quantification existentielle du prédicat avant-après équivaut au prédicat fis, désignant la faisabilité de la substitution (voir le paragraphe 1.3.3) :

$$\exists\, x' \cdot \mathsf{prd}_x\,(S) \Leftrightarrow \mathsf{fis}\,(S)$$

Le prédicat exprimant le domaine d'entrée de l'opération est alors décrit par :

$$\langle \textit{contexte} \rangle \wedge \textit{invariant} \wedge P \wedge \mathsf{fis}\,(S)$$

C'est ce prédicat qui doit faire l'objet de l'analyse de partition. Dans le cadre de nos travaux, nous avons mis l'accent sur la partie spécifique à chaque opération, c'est-à-dire sa partie d'action, spécifiée par la substitution *S*. L'analyse de partition que nous proposons se concentre donc sur fis (*S*) afin de générer les cas de test.

En appliquant l'approche d'analyse de partition définie dans [Dick et Faivre 1993] à des exemples simples de modèles B, nous avons mis en évidence un problème : lors de la réduction sous forme normale disjonctive, on doit faire attention à ne pas générer de termes mal définis. D'une manière informelle, un terme est mal défini quand il contient des fonctions partielles appelées en dehors de leur domaine de définition. Prenons l'exemple du prédicat suivant dans lequel f est une fonction partielle:

$$x \notin \mathrm{dom}(f) \vee y \neq f(x)$$

En utilisant la décomposition en cas disjoints, ce prédicat est décomposé en trois cas suivants :

1. $x \notin \mathrm{dom}(f) \wedge y \neq f(x)$
2. $x \notin \mathrm{dom}(f) \wedge y = f(x)$
3. $x \in \mathrm{dom}(f) \wedge y \neq f(x)$

Les deux premiers cas sont mal définis car l'expression $f(x)$ est dépourvue de sens lorsque la fonction f est appelée en dehors de son domaine. Le problème de la bonne définition des prédicats dans le cadre de la méthode B a été étudié dans [Behm et al. 1998; Burdy 2000]. Ces travaux sont, à notre connaissance, les seuls à avoir abordé B sous cet aspect, et à offrir une solution interfaçable avec l'Atelier B. Le but était d'éliminer les obligations de preuve mal définies. La solution proposée est premièrement, de générer des obligations de preuve supplémentaires pour garantir la bonne définition des modèles, et deuxièmement, de définir un système de déduction approprié. Les auteurs définissent deux opérateurs Δ_p et Δ_s respectivement pour la bonne définition des prédicats et des substitutions. Nous nous basons sur la définition de ces opérateurs. En supposant que les substitutions originales sont bien définies, notre souci est de ne pas générer de termes mal définis pendant la décomposition en cas disjoints. Dans la Figure 4.1, nous donnons un sous-ensemble des règles définissant les opérateurs Δ_p et Δ_s auquel nous nous référerons ultérieurement. Nous rappelons également un théorème qui énonce que l'application d'une substitution bien définie à un prédicat bien défini, est également bien définie.

$$\Delta_p \neg P \equiv \Delta_p P \qquad \Delta_s(P \mid S) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_s S)$$
$$\Delta_p(P \wedge Q) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_p Q) \qquad \Delta_s(S \parallel T) \equiv \Delta_s S \wedge \Delta_s T$$
$$\Delta_p(P \vee Q) \equiv \Delta_p P \wedge (\neg P \Rightarrow \Delta_p Q) \qquad \Delta_s(P \Rightarrow S) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_s S)$$
$$\Delta_p(P \Rightarrow Q) \equiv \Delta_p P \wedge (P \Rightarrow \Delta_p Q) \qquad \Delta_s(S \,?\, T) \equiv \Delta_s S \wedge \Delta_s T$$
$$\Delta_s(@x \cdot S) \equiv \forall x \cdot \Delta_s S$$

Théorème : $\Delta_p I \wedge \Delta_s S \Rightarrow \Delta_p([S]I)$

Figure 4.1. Quelques exemples de définition des opérateurs Δ_p et Δ_s dans [Burdy 2000]

Notons que les règles de définition de Δ_p correspondent à une interprétation des opérateurs logiques de disjonction ($\vee$) et de conjonction ($\wedge$) comme des opérateurs *or else* et *and then,* proposés par certains langages de programmation. En se basant sur

ces définitions, la séparation sûre que nous proposons pour l'opérateur de disjonction ne génère, pour l'exemple précédent, que deux cas disjoints :

1. $x \notin \text{dom}(f)$
2. $x \in \text{dom}(f) \wedge f(x) \neq y$

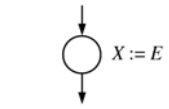
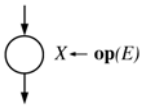
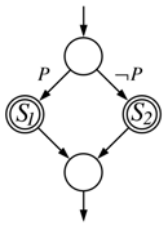
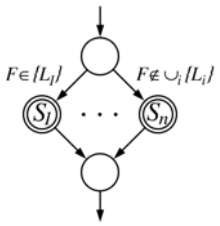
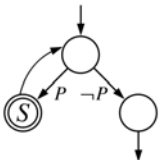
En restreignant la manipulation des prédicats à des manipulations sûres, les analyses de partition définies pour les spécifications Z et VDM peuvent être appliquées aux modèles B obtenus à différentes étapes de développement, car la transformation en prédicat avant-après est définie quel que soit le niveau d'abstraction. L'application de ce type d'approche aux raffinements a été également proposée dans [Dick et Faivre 1993] pour des notations à large spectre comme VDM-SL. Cependant, comme nous l'avons mentionné dans le paragraphe 2.4.1, les approches d'analyse de partition posent des problèmes d'explosion combinatoire. Ces problèmes sont d'autant plus importants que le modèle devient plus détaillé.

4.2.2 Graphe de contrôle des substitutions B0

Dans un composant de type IMPLEMENTATION, seul un sous-ensemble du langage B peut être utilisé. Ce sous-ensemble, appelé B0, ne contient que des substitutions programmables. Nous montrons ces substitutions dans le Tableau 4.1, avec les sous-graphes qui leur sont associés. Notons que l'appel d'opération fait partie du langage considéré, même dans le cas de modèles dépliés : il peut rester des appels non résolus à des opérations vues (lien SEES), pour lesquelles l'environnement de test doit fournir des bouchons.

Les substitutions du B0 étant similaires aux instructions offertes par les langages procéduraux classiques, les sous-graphes adoptés sont également similaires à ceux proposés dans la littérature pour l'analyse structurelle de programmes procéduraux. Ces substitutions peuvent être séquencées par l'opérateur « ; ». Cet opérateur est équivalent au même opérateur « ; » des langages de programmation et signifie que les substitutions reliées se suivent dans l'ordre. En termes de construction du graphe de contrôle, cet opérateur signifie que les sous-graphes correspondant aux substitutions doivent être mis dans l'ordre l'un après l'autre. Ceci se fait facilement en utilisant les sous-graphes du Tableau 4.1 : l'arc sortant d'une substitution est joint à l'arc entrant de la substitution suivante. Pour compléter le graphe de contrôle, on ajoute un nœud de début avant la première substitution, et un nœud de fin après la dernière substitution d'une opération. Le graphe de contrôle construit de cette manière peut éventuellement être simplifié sous une forme plus compacte. Par exemple, un nœud peut regrouper une séquence ininterruptible d'instructions (voir par exemple [Bieman et al. 1998] pour plus d'informations sur la construction du graphe de contrôle des programmes).

Tableau 4.1.Graphe de contrôle des substitutions B0 : X et W sont des (listes de) variables ; E est une (liste de) expression(s) ; F est une expression ; S, S_1, S_2, S_n sont des substitutions ; P est un prédicat ; L_1 est une (liste de) constante(s) ; et ◎ est le sous-graphe correspondant à une substitution

Substitution	Sous graphe	Substitution	Sous graphe
Affectation : $X := E$		Appel d'opération : $X \leftarrow \text{op}(E)$	
Condition IF : IF P THEN S_1 ELSE S_2 END		Condition CASE : CASE F OF EITHER L_1 THEN S_1 OR … ELSE S_n END	
Variable locale: VAR W IN S END		Boucle tant que : WHILE P DO S END	

Les substitutions du B0 sont essentiellement des constructions syntaxiques qui sont proposées pour faciliter l'utilisation du langage. Elles peuvent être traduites en termes de substitutions généralisées de base. Par exemple, la substitution IF est traduite de la manière suivante :

$$(P \Rightarrow \quad S_1) \; ? \; (\neg P \Rightarrow \quad S_2)$$

Nous allons maintenant proposer une réécriture en substitutions de base sous une forme qui permet d'extraire l'expression des chemins du graphe de contrôle.

Comme on peut le constater dans les sous-graphes du Tableau 4.1, l'existence de plusieurs sous-chemins partant d'un nœud correspond à une décision dans la substitution (substitutions IF, CASE, WHILE). En termes de substitutions généralisées de base, ceci correspond à un choix borné entre plusieurs substitutions gardées. Si la substitution originale S est réécrite en $?_i \; S_i$, de sorte que tous les choix aient été « repoussés » à l'extérieur des substitutions S_i, chaque substitution S_i formera un chemin dans le graphe de contrôle. Cette réécriture se fait grâce aux relations d'équivalence suivantes, qui ont été prouvées dans le B-Book [Abrial 1996] :

$$(S_1 \ ? \ S_2) \ ; S_3 = (S_1 \ ; S_3) \ ? \ (S_2 \ ; S_3)$$
$$S_1 \ ; (S_2 \ ? \ S_3) = (S_1 \ ; S_2) \ ? \ (S_1 \ ; S_3)$$
$$@x \cdot (S_1 \ ? \ S_2) = (@x \cdot S_1) \ ? \ (@x \cdot S_2)$$
$$P \Rightarrow \quad (S_1 \ ? \ S_2) = (P \Rightarrow \quad S_1) \ ? \ (P \Rightarrow \quad S_2)$$

à partir desquelles on peut construire quatre règles de réécriture (de la gauche vers la droite). Le graphe de contrôle de la substitution $?_i \ S_i$ ainsi obtenue n'est pas le même que celui de la substitution originale *S*, mais les chemins dans les deux graphes sont les mêmes. Comme d'habitude pour le test des boucles, la substitution WHILE est dépliée un nombre borné *n* de fois. De la sorte, nous ne considérons que les chemins correspondant à 0, 1, …, *n* passages dans la boucle.

D'après la définition de l'opérateur Δ_s donnée dans [Burdy 2000] (voir la Figure 4.1), les quatre règles de réécriture ci-dessus n'introduisent pas de substitutions mal définies. Nous montrons, dans la Figure 4.2, la démonstration que nous avons effectuée pour les règles de réécriture des substitutions gardée et choix non-borné. La démonstration pour les deux règles relatives à la substitution de séquencement est immédiate, car [Burdy 2000] utilisent la distributivité du séquencement « ; » par rapport au choix borné « ? » dans leur définition de Δ_s.

$$\begin{aligned}
\Delta_s (P \Rightarrow \quad (S_1 \quad S_2)) &\Leftrightarrow \Delta_p \, P \wedge (P \Rightarrow \Delta_s (S_1 \quad S_2)) \\
&\Leftrightarrow \Delta_p \, P \wedge (P \Rightarrow (\Delta_s \, S_1 \ \wedge \Delta_s \, S_2)) \\
&\Leftrightarrow \Delta_p \, P \wedge (P \Rightarrow \Delta_s \, S_1 \ \wedge P \Rightarrow \Delta_s \, S_2) \\
&\Leftrightarrow \Delta_s (P \Rightarrow \quad S_1) \wedge \Delta_s (P \Rightarrow \quad S_2) \\
&\Leftrightarrow \Delta_s ((P \Rightarrow \quad S_1) \ \tilde{n} \ (P \Rightarrow \quad S_2)) \\
\\
\Delta_s (@x \cdot (S_1 \ ? \ S_2)) &\Leftrightarrow \forall x \cdot \Delta_s (S_1 \ ? \ S_2) \\
&\Leftrightarrow \forall \, x \cdot (\Delta_s \, S_1 \wedge \Delta_s \, S_2) \\
&\Leftrightarrow (\forall \, x \cdot \Delta_s \, S_1) \wedge (\forall \, x \cdot \Delta_s \, S_2) \\
&\Leftrightarrow \Delta_s (@x \cdot S_1) \wedge \Delta_s (@x \cdot S_2) \\
&\Leftrightarrow \Delta_s ((@x \cdot S_1) \ \tilde{n} \ (@x \cdot S_2))
\end{aligned}$$

Figure 4.2. Bonne définition des règles de réécriture des substitutions gardées et choix non-borné

Tout comme dans le graphe de contrôle des programmes, certains chemins peuvent ne pas être faisables, c'est-à-dire qu'il n'existe aucune attribution de valeurs aux variables d'entrée qui activera ces chemins. Le prédicat fis de la substitution réécrite nous donne les chemins faisables dans le graphe de contrôle :

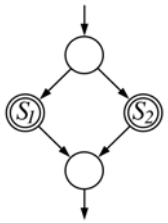
$$\text{fis} (\ _i \, S_i) \Leftrightarrow \vee_i \, \text{fis} \, (S_i)$$

Ceci montre que les techniques de sélection de chemins sur le graphe de contrôle peuvent être exprimées en terme de couverture de la structure propositionnelle du prédicat fis.

4.2.3 Extension de la notion de graphe de contrôle

Le paragraphe précédent ne traitait que des substitutions du B0. Nous allons maintenant proposer une extension de la notion de graphe de contrôle pour prendre en compte les substitutions plus abstraites (voir le Tableau 4.2). Cette extension préserve la possibilité d'identifier les chemins du graphe de contrôle par réécriture de toute substitution S en $?_i\ S_i$. De plus, nous proposons une interprétation opérationnelle de la notion de couverture du graphe. Pour les substitutions B0, comme pour les programmes, une *exécution* avec une entrée de test correspond à la couverture d'un et un seul chemin sur le graphe : le chemin i tel que fis (S_i) est vérifié pour l'entrée de test. Or, ceci n'est plus vrai pour les substitutions abstraites, comme nous le verrons dans les paragraphes suivants.

Tableau 4.2. Graphe de contrôle des substitutions choix borné, parallèles et gardées : S, S_1, S_2 sont des substitutions ; P est un prédicat ; ◎ est le graphe correspondant à une substitution

Substitution	Sous-graphe	Substitution	Sous-graphe
Substitution choix borné : CHOICE S_1 OR S_2 $\equiv S_1\ ?\ S_2$ END		Substitution parallèle : $S_1 \parallel S_2$	
Substitution gardée : SELECT P THEN S $\equiv P \Rightarrow S$ END			

4.2.3.1 Substitution choix borné

La substitution choix borné peut être utilisée à différents niveaux de développement, dans les composants MACHINE, REFINEMENT ou IMPLEMENTATION. Dans le composant IMPLEMENTATION la spécification doit être déterministe. Ainsi, la substitution choix borné est utilisée seulement dans une décision : les différentes substitutions composant le choix sont gardées par des prédicats exclusifs dont la disjonction est vraie (exemple : P et $\neg P$ pour le IF). Dans un composant MACHINE ou REFINEMENT, la substitution choix borné peut être utilisée d'une manière indéterministe : la substitution $S = S_1\ ?\ S_2$ signifie que la substitution S peut être indifféremment implémentée par une implémentation de S_1 ou de S_2.

Pour représenter cette substitution dans le graphe de contrôle, nous proposons le sous-graphe du Tableau 4.2. La substitution choix borné $S = S_1\ ?\ S_2$ a la forme désirée $?_i\ S_i$, chaque S_i désignant un chemin. Comme on peut le voir, ce sous-graphe est

similaire au sous-graphe de la substitution IF de B0, à la différence près que dans celui-ci il n'y a pas de prédicats associés aux arcs entrants des sous-graphes de S_1 et S_2. Les substitutions S_1 et S_2 peuvent ne pas être des substitutions gardées, ou les prédicats des gardes peuvent être non exclusifs : les deux substitutions peuvent alors être faisables en même temps. Il faut donc donner une interprétation opérationnelle à la couverture de ce sous-graphe.

Dans le chapitre 3, dans le contexte du test des modèles B via l'animation (voir le paragraphe 3.5.2), nous avons indiqué que le résultat d'animation d'une spécification indéterministe doit être l'ensemble des résultats possibles. Nous avons justifié cette exigence en nous référant à la notion d'approximation correcte définie dans [Breuer et Bowen 1994], dans le cadre de la définition d'une animation correcte des spécifications Z. De la même manière, pour la couverture du sous-graphe des substitutions indéterministes, nous adoptons l'interprétation opérationnelle suivante : si pour certaines valeurs d'entrée les deux composantes de la substitution choix borné sont faisables, les deux branches du sous-graphe correspondant seront couvertes. Par exemple considérons la substitution $(x > 3 \Rightarrow y := 1)$? $(x \leq 5 \Rightarrow y := 2)$. Pour les valeurs $x > 5$, seule la première substitution est faisable et donc un seul chemin est couvert. Pour les valeurs $x \leq 3$, seule la deuxième substitution est faisable et l'autre chemin du graphe est couvert. Pour les valeurs $x \in \{4, 5\}$, les deux substitutions sont faisables et les deux chemins du graphe sont couverts en même temps.

A cause des substitutions indéterministes qui peuvent exister dans des composants de type MACHINE ou REFINEMENT, une *exécution* (par animation) d'une substitution peut ne pas correspondre à la couverture d'un seul chemin dans le graphe de contrôle, mais à un ensemble de chemins. Nous appelons cet ensemble de chemins, qui sont *exécutables* (faisables) en même temps, une *combinaison de chemins*.

4.2.3.2 Substitution gardée

Mis à part l'indéterminisme, un autre problème que nous devons considérer est la notion de substitution infaisable. La notion de substitution infaisable est liée aux substitutions gardées. Dans le Tableau 4.2, nous proposons un sous-graphe pour la substitution gardée. Comme nous l'avons expliqué dans le Chapitre 1, la substitution gardée $P \Rightarrow S$ indique que la substitution S est faisable seulement si la garde P est vraie. De la même manière, le sous-graphe de la substitution S est couvert seulement quand la garde est vraie. Nous rappelons, du paragraphe 4.2.2, l'équivalence prouvée dans le B-Book, qui permet de distribuer la substitution gardée sur la substitution choix borné :

$$P \Rightarrow (S_1 \,?\, S_2) = (P \Rightarrow S_1) \,?\, (P \Rightarrow S_2)$$

Comme nous l'avons mentionné précédemment, il est possible qu'un composant de type MACHINE ou REFINEMENT contienne des substitutions infaisables. Si un miracle a été spécifié dans un composant de type MACHINE ou REFINEMENT, il peut y avoir des valeurs d'entrée pour lesquelles aucun chemin ne peut être couvert. Ceci est vrai quand les gardes sont fausses. Par exemple, soit la substitution (x < 3 ⇒ y := 1) ? (x ≥ 5 ⇒ y := 2) : pour les valeurs $x \in \{3, 4\}$, les deux substitutions composantes

sont infaisables. À cause de telles substitutions, une combinaison de chemins peut être l'ensemble vide.

4.2.3.3 Substitution parallèle

La substitution parallèle $S = S_1 \parallel S_2$ indique l'exécution simultanée des substitutions S_1 et S_2. Dans ce cas le parallélisme signifie que les deux substitutions, modifiant des variables distinctes, peuvent se réaliser indépendamment. Comme nous l'avons expliqué dans le chapitre 1, la substitution parallèle est une généralisation de la substitution multiple. Ainsi, conformément aux égalités définies dans le B-Book, on peut toujours se ramener à une substitution multiple en distribuant la substitution parallèle sur les autres substitutions. Cependant, la substitution parallèle facilite l'écriture de spécifications complexes et sa distribution modifie considérablement la structure de telles spécifications. Par conséquent, nous avons choisi de proposer un sous-graphe de contrôle pour cette substitution que nous montrons dans le Tableau 4.2.

Une substitution parallèle est faisable si et seulement si toutes ses composantes sont faisables. Nous le visualisons sur le sous-graphe du Tableau 4.2 en connectant les arcs du sous-graphe. Si pour certaines valeurs d'entrée au moins une des substitutions est infaisable, le sous-graphe de la substitution parallèle n'est pas couvert par ces valeurs.

Comme pour les autres substitutions, une substitution S qui contient des substitutions parallèles peut aussi être réécrite sous la forme $?_i\ S_i$ et chaque S_i est un chemin dans le graphe de contrôle. Les équivalences suivantes ont été prouvées dans le B-Book :

$$(S_1 \ ?\ S_2\ ) \parallel S_3 = (S_1 \parallel S_3)\ ?\ (S_2 \parallel S_3)$$
$$S_1 \parallel (S_2\ ?\ S_3\ ) = (S_1 \parallel S_2)\ ?\ (S_1 \parallel S_3)$$

Comme pour les autre règles de réécriture, ces deux règles n'introduisent pas de substitutions mal définies. Dans la Figure 4.3, nous le démontrons pour la première règle, la démonstration pour la deuxième règle procédant de façon similaire.

$$\begin{aligned}
\Delta_s\ ((S_1\ ?\ S_2\ ) \parallel S_3) &\Leftrightarrow \Delta_s\ (S_1 \quad S_2) \wedge \Delta_s\ S_3 \\
&\Leftrightarrow (\Delta_s\ S_1\ \wedge \Delta_s\ S_2) \wedge \Delta_s\ S_3 \\
&\Leftrightarrow (\Delta_s\ S_1\ \wedge \Delta_s\ S_3) \wedge (\Delta_s\ S_2 \wedge \Delta_s\ S_3) \\
&\Leftrightarrow \Delta_s\ (S_1 \parallel S_3) \wedge \Delta_s\ (S_2 \parallel S_3) \\
&\Leftrightarrow \Delta_s\ ((S_1 \parallel S_3)\ \tilde{n}\ (S_2 \parallel S_3))
\end{aligned}$$

Figure 4.3. Bonne définition d'une des règles de réécriture de la substitution parallèle

Pour les substitutions parallèles, la notion de chemin ne correspond plus à la définition classique de chemin dans un graphe. Par exemple, considérons le graphe de contrôle de la Figure 4.4 : nous lui associons deux chemins et non pas trois. Ce sont les chemins que nous obtenons après la réécriture de la substitution originale.

Substitution originale :

$S_1 \parallel (S_2 \; ? \; S_3)$

Substitution réécrite :

$(S_1 \parallel S_2) \; ? \; (S_1 \parallel S_3)$

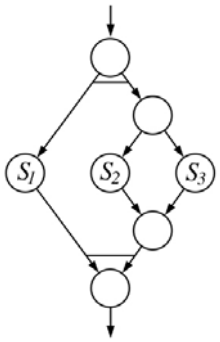


Figure 4.4. Exemple d'un graphe de contrôle pour une substitution parallèle. Les chemins dans le graphe sont $S_1 \parallel S_2$ et $S_1 \parallel S_3$.

4.2.3.4 Substitution pré-conditionnée

La dernière substitution généralisée de base que nous devons considérer est la substitution pré-conditionnée. En fait, dans la construction du graphe de contrôle, nous supposons que la pré-condition est vérifiée, de même que l'invariant et le contexte de la machine abstraite. Ces hypothèses, que nous avons également considérées pour l'analyse des prédicats avant-après (voir le paragraphe 4.2.1), sont nécessaires pour que le texte formel de l'opération ait un sens. D'une part on a besoin des déclarations des entités manipulées par l'opération, et d'autre part ces entités doivent vérifier certaines propriétés pour que l'on puisse raisonner en terme de faisabilité. Par exemple, considérons la faisabilité de la substitution $P \mid S$:

$$\mathsf{fis}\,(P \mid S) \Leftrightarrow P \Rightarrow \mathsf{fis}\,(S)$$

D'après [Burdy 2000], S et donc $\mathsf{fis}\,(S)$, peuvent être mal définis si la pré-condition n'est pas vérifiée (voir la Figure 4.1). Rappelons également que dans le paragraphe 3.5.2.2, nous avons exigé que l'outil d'animation refuse de calculer de valeurs de sortie lorsqu'une opération est appelée en dehors de sa pré-condition.

La substitution pré-conditionnée n'a donc pas de représentation spécifique en sous-graphe de contrôle. Soit la pré-condition P est vérifiée, et dans ce cas le graphe de contrôle de la substitution est celui de S, soit P n'est pas vérifiée, et dans ce cas la notion de graphe de contrôle pour la substitution est dépourvue de sens.

4.2.4 Couverture du graphe de contrôle et des prédicats avant-après

Dans ce paragraphe, nous résumons les concepts développés jusqu'ici, et établissons un lien entre la couverture du graphe de contrôle et la couverture des prédicats avant-après.

Le domaine d'entrée d'une opération ayant la forme générale PRE *P* THEN *S* END, est exprimé par :

$$<\textit{contexte}> \wedge \textit{invariant} \wedge P \wedge \mathsf{fis}\ (S)$$

Pour générer des cas de test, nous pouvons décomposer fis (*S*) en sous-prédicats, chaque sous-prédicat spécifiant un sous-domaine d'entrée à couvrir. Pour cela, des règles de décomposition travaillant avec la structure propositionnelle de fis (*S*) peuvent être utilisées. Cette méthode correspond aux approches de test proposées dans la littérature pour les spécifications Z et VDM.

Toute substitution *S* peut être réécrite en $?_i\ S_i$, en « sortant » la substitution choix borné à l'extérieur des substitutions S_i. Puisque :

$$\mathsf{fis}\ (S) \Leftrightarrow \mathsf{fis}\ (?_i\ S_i) \Leftrightarrow \vee_i\ \mathsf{fis}\ (S_i)$$

la réécriture introduit une première décomposition de fis (*S*) en sous-prédicats. Cette décomposition a une contrepartie sur les graphes de contrôle que nous avons proposés pour les substitutions généralisées : chaque fis (S_i) correspond au sous-domaine d'entrée d'un chemin sur le graphe de contrôle. Cela signifie que la couverture des chemins du graphe de contrôle est équivalente à la couverture d'une partition de fis (*S*) en cas éventuellement non disjoints.

Dans le paragraphe suivant, nous introduirons d'autres règles de décomposition liées au traitement de l'opérateur de disjonction logique dans le prédicat fis (*S*). Comme nous le verrons, les disjonctions apparaissant dans fis (*S*) peuvent venir soit de la structure des substitutions généralisées, soit des prédicats de garde. Nous donnerons, pour chaque cas identifié, une double interprétation en termes de couverture du graphe de contrôle et de couverture de la structure de fis (*S*), faisant ainsi le lien avec les approches proposées dans la littérature. Nos règles de décompositions définissent un ensemble de critères de couverture de sévérité croissante.

4.3 Critères de couverture des substitutions généralisées

Avant de donner la définition de nos critères de couverture, nous allons introduire un exemple simple qui sera utilisé pour illustrer les cas de test engendrés par chaque critère. Nous verrons ensuite comment ces critères peuvent être ordonnés selon leur sévérité.

4.3.1 Préliminaires

L'exemple de la Figure 4.5 montre une substitution généralisée avec le graphe de contrôle qui lui est associé. Nous supposons que la conjonction du contexte, de l'invariant et de la pré-condition de la substitution est exprimée par le prédicat suivant :

$$n_1 \in \text{NAT} \wedge n_2 \in \text{NAT} \wedge special\_n \in \text{BOOL}$$

Cette spécification est d'abord traduite en substitutions de base :

$$S = (@ww \cdot ww \in 0..4 \Rightarrow \quad n_1 := ww) \; ? \\ (special\_n = \text{TRUE} \Rightarrow \quad (((n_1 = 0 \vee n_1 = 1) \Rightarrow \quad n_1 := n_1 + 1) \parallel \\ (n_2 > n_1 \Rightarrow \quad n_2 := n_1)))$$

Nous désignons par S_1 et S_2 chaque composante de la substitution choix borné. Cette substitution est déjà sous la forme S_1 ? S_2 et n'a pas à être réécrite. Les substitutions S_1 et S_2 correspondent aux deux chemins du graphe de contrôle. Le prédicat fis (S) est :

$$\text{fis}(S) \Leftrightarrow \text{fis}(@ww \cdot ww \in 0..4 \Rightarrow \quad n_1 := ww) \vee \\ \text{fis} \quad (special\_n \quad = \quad \text{TRUE}(((n_1 = 0 \vee n_1 = 1) \Rightarrow \quad n_1 := n_1 + 1) \parallel \\ \Rightarrow \\ (n_2 > n_1 \Rightarrow \quad n_2 := n_1))) \\ \Leftrightarrow \text{fis}(S_1) \vee \text{fis}(S_2)$$

```
CHOICE
    ANY ww WHERE ww ∈ 0…4
    THEN n1 := ww
    END
OR
    SELECT special_n = TRUE THEN
        SELECT n1 = 0 ∨ n1 = 1 THEN
            n1 := n1 + 1
        END
        ||
        SELECT n2 > n1 THEN
            n2 := n1
        END
    END
END
```

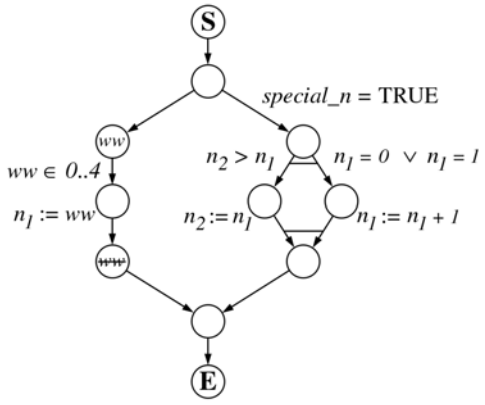


Figure 4.5. Graphe de contrôle construit à partir d'une substitution. La substitution est exécutée dans le contexte suivant : $n_1 \in \text{NAT} \wedge n_2 \in \text{NAT} \wedge special\_n \in \text{BOOL}$

4.3.2 Définition des critères

Dans ce paragraphe, nous allons définir des critères de couverture pour le graphe de contrôle construit à partir d'une substitution généralisée. Chaque critère est défini par une décomposition du prédicat fis (S), liée à l'analyse du flot de contrôle des substitutions.

4.3.2.1 Critère *tous les chemins*

Nous avons vu que toute substitution S peut être réécrite sous la forme $?_i\ S_i$, chaque S_i étant un chemin dans le graphe de contrôle. Nous avons vu également que cette réécriture correspond à une première décomposition de fis (S) en cas éventuellement non-disjoints :

$$\text{fis}\ (S) \Leftrightarrow \text{fis}\ (?_i\ S_i) \Leftrightarrow \vee_i\ \text{fis}\ (S_i)$$

Notre premier critère exige de couvrir chacune des disjonctions ainsi obtenus : les entrées de test doivent alors être générées de manière à satisfaire au moins une fois chaque prédicat fis (S_i) qui n'est pas équivalent à faux. Ce critère est analogue au critère classique *tous les chemins*, défini pour les programmes, car il exige la couverture de tous les chemins faisables du graphe de contrôle.

Dans notre exemple, pour satisfaire ce critère nous devons générer des entrées de test satisfaisant les deux prédicats suivants, qui correspondent à la faisabilité des deux chemins du graphe de contrôle :

$$\text{fis}\ (S_1) \Leftrightarrow \exists\ ww \cdot ww \in 0\ldots4$$
$$\text{fis}\ (S_2) \Leftrightarrow special\_n = \text{TRUE} \wedge (n_1 = 0 \vee n_1 = 1) \wedge n_2 > n_1$$

Le premier chemin est toujours faisable (fis (S_1) est toujours vraie). Pour couvrir le second chemin en même temps que le premier, une seule entrée de test suffit, par exemple $special\_n = \text{TRUE} \wedge n_1 = 0 \wedge n_2 = 1$.

4.3.2.2 Critère *toutes les combinaisons de chemins*

Notre deuxième critère propose de décomposer le prédicat fis (S) $\Leftrightarrow \vee_i$ fis (S_i) en cas disjoints. La décomposition en cas disjoint se fait selon le traitement de l'opérateur de disjonction $\vee$ proposé par [Dick et Faivre 1993], mais sans rentrer dans la structure des fis (S_i). Les prédicats ainsi générés désignent des combinaisons de chemins faisables sur le graphe de contrôle. Nous appelons alors ce critère *toutes les combinaisons de chemins*.

Pour expliquer la décomposition en cas disjoints, considérons l'exemple d'une substitution qui a été réécrite en $S_1\ ?\ S_2$. Le prédicat fis de cette substitution est fis (S_1) $\vee$ fis (S_2). Si S_1 et S_2 sont bien définies, les prédicats fis (S_1) et fis (S_2) le sont aussi (voir le théorème mentionné dans la Figure 4.1, défini dans [Burdy 2000]). Le prédicat fis (S_1) $\vee$ fis (S_2) peut alors être décomposé en trois cas disjoints, sans que des termes mal définis soient générés :

1. fis (S_1) $\wedge$ fis (S_2)
2. $\neg$fis (S_1) $\wedge$ fis (S_2)
3. fis (S_1) $\wedge \neg$fis (S_2)

Le premier cas exige que des entrées de test soient générées pour que S_1 et S_2 soient faisables en même temps. Dans le deuxième et le troisième cas, les entrées de test doivent être générées de manière à ce qu'un des chemins ne soit pas faisable.

Pour notre exemple, les cas disjoints suivants sont générés :

$$\mathsf{fis}\ (S_1) \wedge \mathsf{fis}\ (S_2) \Leftrightarrow special\_n = \text{TRUE} \wedge (n_1 = 0 \vee n_1 = 1) \wedge n_2 > n_1$$
$$\neg\mathsf{fis}\ (S_1) \wedge \mathsf{fis}\ (S_2) \quad \text{Cas impossible à satisfaire}$$
$$\mathsf{fis}\ (S_1) \wedge \neg\mathsf{fis}\ (S_2) \Leftrightarrow \neg(special\_n = \text{TRUE} \wedge (n_1 = 0 \vee n_1 = 1) \wedge n_2 > n_1)$$

Pour couvrir les prédicats ci-dessus, deux entrées de test seront suffisantes. Par exemple, nous pouvons compléter l'entrée de test $special\_n = \text{TRUE} \wedge n_1 = 0 \wedge n_2 = 1$ par $special\_n = \text{FALSE}$.

La décomposition du prédicat $\vee_i\ \mathsf{fis}\ (S_i)$ génère au plus $2^i - 1$ cas. Ceci est une borne supérieure car certains cas peuvent être simplifiés à faux. C'est le cas dans notre exemple dans lequel la substitution S_1 est toujours faisable (donc $\neg\mathsf{fis}\ (S_1)$ est impossible à satisfaire). Pour les opérations d'un composant IMPLEMENTATION, nous ne pouvons pas avoir de combinaisons avec plusieurs chemins faisables : en effet, les opérations de ces composants sont déterministes et implémentables. Pour ces opérations, le critère *toutes les combinaisons de chemins* est équivalent au critère *tous les chemins*.

4.3.2.3 Critères *toutes les combinaisons étendues de chemins*

Dans le critère précédent nous avons considéré des combinaisons de chemins dans lesquelles certains chemins du graphe de contrôle étaient infaisables. Les négations ainsi introduites peuvent engendrer des disjonctions dans le prédicat $\mathsf{fis}\ (S)$, qui peuvent être décomposées. Nous définissons alors notre troisième critère, appelé critère *toutes les combinaisons étendues de chemins*. Si dans une combinaison de chemins, il existe des chemins infaisables, ce critère exige que différents cas pour lesquels ces chemins sont infaisables soient considérés séparément. Ces cas concernent les substitutions gardée et parallèle, que nous considérons par la suite.

La négation de la faisabilité de la substitution gardée est décrite par le prédicat suivant :

$$\neg\mathsf{fis}\ (P \Rightarrow S) \Leftrightarrow \neg(P \wedge \mathsf{fis}\ (S)) \Leftrightarrow \neg P \vee \neg\mathsf{fis}\ (S)$$

Ce prédicat signifie que la substitution gardée est infaisable si la garde P est fausse ou si la substitution S est infaisable. Il peut être décomposé en cas disjoints. Cependant, pour ce prédicat, nous ne pouvons pas utiliser la décomposition en cas disjoints proposée par [Dick et Faivre 1993]. D'après la définition de l'opérateur Δ_s donnée dans [Burdy 2000] (voir la Figure 4.1) pour les substitutions gardées, la substitution composante d'une substitution gardée peut être mal définie si le prédicat de la garde est faux. Pour ne pas générer de termes mal définis, nous utilisons une décomposition sûre en cas disjoints :

$\neg\mathsf{fis}\ (P \Rightarrow S) \Leftrightarrow \neg(P \wedge \mathsf{fis}\ (S))$ est décomposé en : 1. $\neg P$

2. $P \wedge \neg\mathsf{fis}\ (S)$

La négation de la faisabilité de la substitution parallèle est décrite par le prédicat suivant :

$$\neg\mathsf{fis}\ (S \parallel T) \Leftrightarrow \neg(\mathsf{fis}\ (S) \wedge \mathsf{fis}\ (T)) \Leftrightarrow \neg\ \mathsf{fis}\ (S) \vee \neg\mathsf{fis}\ (T)$$

L'opérateur Δ_s de la substitution parallèle défini dans [Burdy 2000] précise que si la substitution parallèle est bien définie, alors les deux substitutions composantes le sont également. Pour la substitution parallèle, la négation du prédicat fis peut donc être décomposée en trois cas disjoints :

$\neg$fis (S || T) $\Leftrightarrow$ $\neg$(fis (S) $\wedge$ fis (T)) est décomposé en :
1. $\neg$fis (S) $\wedge$ $\neg$fis (T)
2. $\neg$fis (S) $\wedge$ fis (T)
3. fis (S) $\wedge$ $\neg$fis (T)

Les décompositions de la négation des substitutions gardée et parallèle contiennent à leur tour des négations sur la faisabilité des substitutions composantes. Si ces négations génèrent des disjonctions dans le prédicat fis, ces disjonctions seront également décomposées de manière récursive, en suivant les décompositions ci-dessus.

Dans notre exemple, dans le dernier prédicat généré par le critère *toutes les combinaisons de chemins*, la substitution S_2 est infaisable. Chaque composante de S_2, qui peut rendre la substitution infaisable, sera considérée séparément. Nous devons compléter les entrées de test précédentes pour satisfaire les prédicats suivants :

$special\_n$ = FALSE
$special\_n$ = TRUE $\wedge$ $\neg(n_1 = 0 \vee n_1 = 1)$ $\wedge$ $\neg(n_2 > n_1)$
$special\_n$ = TRUE $\wedge$ $(n_1 = 0 \vee n_1 = 1)$ $\wedge$ $\neg(n_2 > n_1)$
$special\_n$ = TRUE $\wedge$ $\neg(n_1 = 0 \vee n_1 = 1)$ $\wedge$ $(n_2 > n_1)$

Les cas ci-dessus résultent de la décomposition sûre de la substitution gardée et de la substitution parallèle en cas disjoints.

4.3.2.4 Couverture des prédicats de gardes

Dans les critères précédents, nous avons considéré la structure des opérateurs du langage des substitutions généralisées, et avons proposé différentes décompositions en cas de test. Nous pouvons également considérer la structure des prédicats de garde. Pour ces prédicats, nous proposons la décomposition suivante en cas disjoints pour les opérateurs logiques de disjonction et d'implication et pour la négation de l'opérateur de conjonction :

$A \vee B$ est décomposé en : 1. A 2. $\neg$A $\wedge$ B
$A \Rightarrow B$ est décomposé en : 1. $\neg A$ 2. $A \wedge B$
$\neg(A \wedge B)$ est décomposé en : 1. $\neg A$ 2. $A \wedge \neg B$

Tous les critères précédents liés à la sélection de chemins sur le graphe de contrôle peuvent donc être raffinés en exigeant que chaque prédicat de garde soit décomposé en cas disjoints au moins une fois pendant le test. Par exemple, le critère *tous les chemins* plus la couverture des prédicats exige que tous les chemins faisables dans le graphe de contrôle soient couverts et que tous les prédicats de garde soient décomposés en cas disjoints au moins une fois. Dans notre exemple, pour couvrir le critère tous les chemins, nous devions générer des entrées de test satisfaisant le prédicat :

$$\mathsf{fis}\ (S_2) \Leftrightarrow special\_n = \text{TRUE} \wedge (n_1 = 0 \vee n_1 = 1) \wedge n_2 > n_1$$

Pour couvrir également les prédicats de garde, le prédicat $(n_1 = 0 \vee n_1 = 1)$ est séparé en cas disjoints. Pour cela, l'entrée de test $special\_n = \text{TRUE} \wedge n_1 = 0 \wedge n_2 = 1$ peut être complétée par $special\_n = \text{TRUE} \wedge n_1 = 1 \wedge n_2 = 2$.

Les décompositions ci-dessus sont des décompositions sûres qui ne généreront pas de termes mal définis. Elles sont basées sur la définition de l'opérateur Δ_p pour la bonne définition des prédicats [Burdy 2000]. Il existe d'autres critères de couverture de prédicats qui ont été proposés dans la littérature pour la couverture structurelle des décisions dans les programmes (voir par exemple [Chilenski et Miller 1994] dans le domaine avionique). Ces critères peuvent également être utilisés, sous réserve qu'ils puissent être adaptés aux règles de bonne définition de la notation B.

4.3.3 Hiérarchie des critères

La hiérarchie des critères que nous avons définis dans le paragraphe précédent est montrée dans la Figure 4.6. La relation entre les critères est la relation d'*inclusion* [Frankl et Weyuker 1988]. La relation d'inclusion pour nos critères est facilement obtenue d'après leur définition. Tous nos critères sont définis à partir d'une décomposition plus ou moins fine de l'opérateur de disjonction logique dans le prédicat fis. Le critère *toutes les combinaisons de chemins* décompose le prédicat $\vee_i$ fis (S_i) en cas disjoints, générant ainsi toutes les combinaisons de chemins faisables du graphe de contrôle. Il inclut alors le critère *tous les chemins* qui exige seulement la couverture de chaque fis (S_i), correspondant aux chemins faisables du graphe de contrôle. Le critère *toutes les combinaisons étendues de chemins* exige une décomposition encore plus fine du prédicat fis et inclut ainsi le critère *toutes les combinaisons de chemins*. Chacun de ces critères est inclus par le même critère plus la couverture des prédicats de garde, car celui-ci exige que chaque prédicat soit décomposé au moins une fois pendant le test.

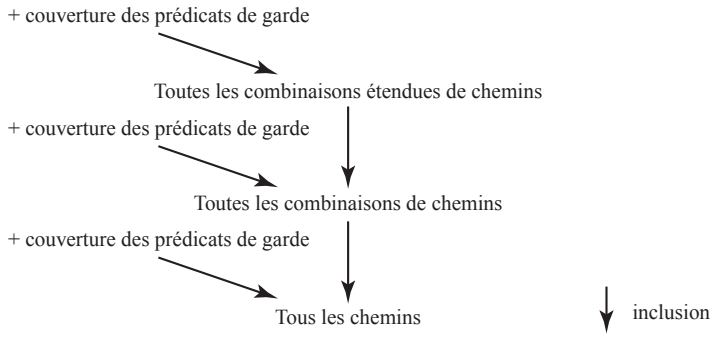


Figure 4.6. Hiérarchie des critères

Ces critères sont définis pour couvrir les spécifications à différents niveaux d'abstraction, qui peuvent contenir des constructions indéterministes et infaisables. Sous le critère *tous les chemins*, nous retrouvons les critères de couverture structurelles classiques qui ont été définis pour les programmes (par exemple : *toutes les branches*, …).

Nous pouvons également comparer ces critères avec l'approche proposée dans [Dick et Faivre 1993]. Les cas que nous avons identifiés à partir des structures des substitutions et des prédicats de garde correspondent à une décomposition des disjonctions dans le prédicat fis. Le critère de [Dick et Faivre 1993] est donc au moins aussi sévère que le critère *toutes les combinaisons étendues de chemins* plus la couverture des prédicats de garde. En fait, nous allons montrer qu'il est plus sévère.

Remarquons tout d'abord que, dans nos critères, la décomposition de chaque prédicat de garde n'est exigée qu'une seule fois, ce qui n'est pas le cas pour [Dick et Faivre 1993]. Dans notre terminologie, cela signifie que [Dick et Faivre 1993] exigent que chaque combinaison étendue de chemin soit testée avec toutes les combinaisons possibles de cas issus de la décomposition des prédicats de garde.

De plus, le critère de [Dick et Faivre 1993] peut générer des cas de test qui partitionnent le domaine de sortie. Considérons l'exemple suivant, ou x est un paramètre d'entrée et y un paramètre de sortie :

IF $x > 0$ THEN $y := x$ ELSE $y := 1$ END

En transformant cette substitution en prédicat avant-après, et en appliquant l'approche de [Dick et Faivre 1993], trois sous-prédicats suivants sont générés :

1. $x \leq 0 \wedge y \neq x \wedge y = 1$
2. $x > 0 \wedge y = x \wedge y \neq 1$
3. $x > 0 \wedge y = x \wedge y = 1$

Une quantification existentielle sur le paramètre de sortie y génère les trois cas de test suivant :

1. $x \leq 0$
2. $x > 1$
3. $x = 1$

Pour la première branche du IF, c'est-à-dire quand $x > 0$, deux cas sont engendrés : le cas $x > 1$ exige que la valeur de sortie soit différente de celle qui aurait été obtenue par l'autre branche du IF ; le cas $x = 1$ exige que le résultat soit identique, c'est-à-dire $y = 1$. La distinction entre ces cas est le résultat d'une partition du domaine de sortie. Dans nos critères, en travaillant directement sur le prédicat fis au lieu du prédicat avant-après, nous obtenons une partition du domaine d'entrée uniquement. Par exemple, lors de la décomposition de fis (S_1 ? S_2) en fis (S_1) $\vee$ fis (S_2), le quantificateur existentiel sur les variables de sortie est « descendu » au niveau de chaque S_i. Les seules variables communes sont donc des variables d'entrée.

L'approche proposée dans [Dick et Faivre 1993] pose des problèmes d'explosion combinatoire. Même pour une spécification relativement simple, un nombre important

de cas de test est généré. Comme notre analyse procède à partir des substitutions, et non de leur transformation en prédicats avant-après, il a été possible de distinguer les cas qui sont générés à partir de la structure des opérateurs du langage des substitutions généralisées, et les cas correspondant à la structure propositionnelle des prédicats de garde. Ceci nous a permis de définir des critères avec différents degrés de sévérité, de manière à ce que la stratégie de couverture puisse être adaptée à la complexité de l'opération. De plus, notre analyse structurelle prend en compte les règles de bonne définition des substitutions et des prédicats, et ne peut générer de cas dépourvus de sens.

4.4 Discussion

Une première analyse des opérations spécifiées dans l'étude de cas que nous avons mentionnée dans le Chapitre 3, nous a permis de tirer quelques conclusions sur la méthode d'analyse structurelle que nous avons proposée. Nous avons regardé différentes opérations offertes par des sous-systèmes du logiciel, à la fois au niveau abstrait et au niveau concret, pour identifier des critères de test applicables.

Le style de programmation adopté dans cette étude de cas suit les grandes lignes suivantes :

- Dans les composants MACHINE et les composants REFINEMENT éventuels, les opérations spécifient les variables d'état qui peuvent être modifiées, la valeur que ces variables peuvent prendre étant exprimée sous forme de post-condition (substitution de type $x :P$, voir le Tableau 1.2). Pour ces opérations, la complexité de la spécification vient de la structure propositionnelle des post-conditions (qui sont donc des prédicats de garde).
- Dans les composants IMPLEMENTATION, les opérations sont spécifiées d'une manière algorithmique en utilisant des constructions programmables, par exemple des substitutions IF ou CASE. Dans ce cas, la complexité de la spécification vient essentiellement de la structure de ces substitutions.

Pour les niveaux abstraits, c'est-à-dire pour les opérations des composants MACHINE ou REFINEMENT, les cas de test doivent être extraits à partir des prédicats de post-condition. Etant donnée la complexité de ces prédicats, une analyse fine de ceux-ci, comme par exemple la décomposition proposée par [Dick et Faivre 1993], et adaptée aux règles de bonne définition de B, est impraticable. Des critères moins exigeants doivent être envisagés. Par exemple, pour un prédicat de forme $\wedge_i (A_i \vee B_i)$, $1 \leq i \leq n$, on considèrera séparément chaque disjonction, sans chercher à couvrir des combinaisons de cas sur les i disjonctions. Pour les niveaux concrets, c'est-à-dire pour les opérations des composants IMPLEMENTATION, les critères applicables aux programmes par exemple le critère *tous les chemins*, peuvent être envisagés. Cependant, pour les opérations dépliées qui traversent des niveaux de décomposition, le graphe de contrôle devient vite complexe. Ceci exige l'application de critères plus

faibles, comme le critère *toutes les branches*, éventuellement complétés par une couverture peu sévère des prédicats de garde.

Reprenons l'exemple du sous-système que nous avons présenté dans le Chapitre 3. Nous rappelons l'architecture de ce sous-système dans la Figure 4.7. La spécification de l'opération offerte par ce sous-système dans le composant SOUS_SYSTEME.mch n'est constituée que du typage des variables. Dans le composant SOUS_SYSTEME_IMP.imp, le raffinement de l'opération est constituée d'une séquence d'appels aux opérations des composants importés A_NIVEAU_1.mch, B_NIVEAU_1.mch et C_NIVEAU_1.mch. Dans ce qui suit, nous appelons respectivement sous-système A, B, C les modèles dépliés (en grisé sur la figure) dont ces trois composants sont la racine.

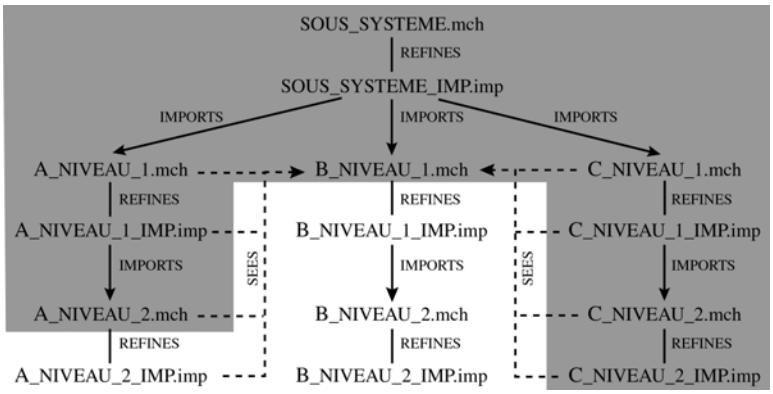


Figure 4.7. Architecture symbolique d'un des sous-systèmes de l'étude de cas

La version dépliée des opérations des sous-systèmes A et B reste à un niveau abstrait. Dans chaque cas, la complexité vient de la structure propositionnelle des prédicats des post-conditions (conjonction de disjonctions et d'implications imbriquées). La version dépliée de l'opération du sous-système C est plus concrète. Elle est constituée d'une séquence de substitutions IF et CASE, correspondant aux corps des opérations implémentées dans le composant C_NIVEAU_2.imp. Cette structure dépliée est complexe (24000 chemins syntaxiques dans le graphe de contrôle), empêchant l'application du critère structurel *tous les chemins*. Le seul critère structurel applicable est le critère *toutes les branches* (48 branches) complété par un critère non-exigeant de couverture des prédicats.

Les critères structurels *toutes les combinaisons de chemins* et *toutes les combinaisons étendues de chemins* s'appliquent essentiellement aux spécifications indéterministes faisant un usage combiné de substitutions choix borné, parallèle et gardée. Dans cette étude de cas, nous n'avons pas pu trouver des exemples de telles spécifications. Cependant, ce style de spécification peut être trouvé dans d'autres études de cas du domaine public [Hoare 1995; Taouil-Traverson 1997].

L'étude de cas fournie par Alstom Transport SA nous a permis de confirmer que, pour un exemple réel de développement en B, nous atteignons vite les limites des approches structurelles. Les seuls critères applicables sont des critères faibles : ceci montre l'intérêt de disposer d'une hiérarchie de critères de différents degrés de sévérité, mais pose le problème de la pertinence de l'information structurelle vis-à-vis des fautes que l'on cherche à révéler. L'analyse manuelle de la structure des modèles est fastidieuse, voire impossible à réaliser dans le cas d'un modèle du système complet. Il reste à déterminer dans quelle mesure l'automatisation de cette analyse nous permettra de traiter des modèles de taille réaliste.

Par analogie avec le test de programme, on peut envisager des stratégies de test progressives, basées sur un processus d'intégration. En définissant la notion d'étape de développement pour des sous-systèmes (chapitre 3), nous avons voulu laisser ouverte cette possibilité. Le processus d'intégration peut procéder selon une approche ascendante, en remontant les liens d'importation. Dans l'exemple de la Figure 4.7, cela reviendrait à tester d'abord les modèles dépliés des sous-systèmes A, B, et C, avant de tester le modèle du sous-système complet, ce dernier étant ensuite intégré au modèle complet de l'application. On peut également envisager une intégration descendante, qui suit les étapes de développement. Dans l'exemple de la Figure 4.7, cela reviendrait à d'abord tester le modèle de racine SOUS_SYSTEME.mch et de feuilles A_NIVEAU_1.mch, B_NIVEAU_1.mch, C_NIVEAU_1.mch, puis à tester des raffinements de ce modèle. Quelle que soit l'approche d'intégration envisagée, il faudra définir comment exploiter les liens structurels entre modèles pour prendre en compte, à une étape donnée, ce qui a été testé aux étapes précédentes.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une approche *unifiée* pour la couverture structurelle de modèles B : elle peut être appliquée aux modèles qui contiennent à la fois des constructions abstraites et concrètes. Cette approche a été définie en poursuivant notre cadre unifié du test, dans lequel des modèles dépliés, obtenus aux différentes étapes de développement, font l'objet du test.

Nous avons proposé une extension à la notion classique de graphe de contrôle, qui peut être appliquée aux substitutions abstraites de la notation B. Le graphe de contrôle offre une vision synthétique de la structure des opérations d'un modèle B. Dans notre cas, cette information est destinée à être exploitée pour le test, mais nos travaux peuvent être utilisés dans un cadre plus général, pour l'analyse structurelle des modèles B.

Nous avons défini un ensemble de critères de couverture du graphe de contrôle, partiellement ordonnés selon la relation d'inclusion, qui viennent compléter la hiérarchie de critères existants. Ces critères distinguent les cas de test générés à partir de la structure des substitutions généralisées, et les cas générés à partir de la structure des prédicats de garde. L'objectif est de permettre de choisir le critère à appliquer

selon la complexité de la spécification analysée. De plus, dans la définition de nos critères nous avons pris en compte la bonne définition des prédicats et des substitutions. Ainsi, les cas de test générés ne contiennent pas de termes mal définis.

Pour la couverture des prédicats de garde, nous avons considéré uniquement la structure propositionnelle de ces prédicats. Cette couverture peut être complétée par la couverture des opérateurs de la théorie des ensembles. Un exemple de ce type de traitement, que nous avons également mentionné dans le chapitre 2, est la décomposition du prédicat $e \in A_1 \cup A_2$ en trois cas :

1) $e \in A_1 \cap A_2$ 2) $e \in A_1 - A_2$ 3) $e \in A_2 - A_1$

On peut également envisager une couverture des valeurs aux limites, par exemple en générant les cas $x = -1$, $x = 0$, $x = 1$ à partir de l'analyse du prédicat $x \leq 0$.

Comme nous l'avons mentionné au début de ce chapitre, l'approche présentée concerne uniquement l'analyse des opérations d'un modèle B. Les enchaînements possibles des opérations doivent également être pris en compte afin de générer des séquences de test. Pour ce faire, l'approche proposée dans la littérature, que nous avons mentionnée dans le paragraphe 2.4.1.2, consiste à couvrir un automate à états finis construit à partir de la spécification. La possibilité d'appliquer cette approche à des modèles B réalistes, au moins à ceux obtenus lors des premières phases de développement, doit être étudiée.

Les cas de test générés par nos critères impliquent la résolution des prédicats de la logique du premier ordre. La logique du premier ordre étant semi-décidable, cette démarche ne peut pas être entièrement automatisable. Une approche alternative peut être d'utiliser d'autres moyens de génération des séquences de test (par exemple, à partir d'une analyse des besoins fonctionnels), et d'évaluer a posteriori la couverture du graphe de contrôle fournie par les séquences de test générées. Cette évaluation peut se faire par instrumentation du modèle B testé, afin d'obtenir une trace de l'exécution de l'opération. Par exemple, en ajoutant des substitutions simples dans le source des spécifications, les nœuds du graphe qui sont traversés pendant une exécution peuvent être marqués. Cependant, l'analyse de couverture de la trace ainsi obtenue devra être formalisée, pour tenir compte des exécutions parallèles et indéterministes.

Conclusion générale

Nous avons proposé dans ce mémoire un cadre théorique de test adapté au processus de développement formel B. Le test a pour but de révéler les fautes dans la spécification et visent à compléter les preuves prescrites par la méthode B. Ces preuves garantissent la cohérence mathématique des modèles et la préservation de la sémantique de ceux-ci par les raffinements. Ce processus de vérification interne ne garantit en rien la correction de la spécification elle-même.

La littérature a souligné l'importance de la vérification de la spécification vis-à-vis des besoins de l'utilisateur et ceci dès les phases amont du développement. Les spécifications exécutables permettent d'utiliser des techniques de vérification dynamique plus tôt dans le développement, afin de valider la spécification. Nous avons défini un cadre unifié de test, indépendant du fait que les résultats de test soient obtenus par animation de modèles B ou par exécution du code final. Les deux outils commerciaux de développement en B disposent chacun d'un outil d'animation. Nous avons cependant montré que ces outils d'animation sont insuffisants pour le cadre théorique proposé, notamment en ce qui concerne le traitement de l'indéterminisme de la spécification. Nous avons pu trouver dans la littérature des travaux relatifs à la notation Z, qui permettent de placer nos exigences dans le contexte plus général de la spécification d'outils d'animation. Ces travaux proposent une formalisation de la notion d'interprétation exécutable correcte de modèles qui couvre les besoins que nous avons identifiés pour les cas indéterministes. Nous avons mentionné également des travaux académiques visant à améliorer les techniques d'animation de spécifications B. Nous espérons que ces travaux pourront proposer une solution satisfaisante.

La notion d'oracle de test est définie en liaison avec celle de raffinement, de sorte que si, à une phase du développement, l'animation de modèles fournit des résultats corrects, alors les obligations de preuves de B garantissent que tout raffinement ultérieur fournira aussi des résultats corrects pour les mêmes entrées de test. L'identification des phases du développement est basée sur la notion de dépliage de

modèles, qui permet de raisonner formellement sur le comportement observable induit par une architecture de composants B impliquant différents niveaux de raffinement. Un exemple simple (le Triangle) a permis de montrer que la phase à partir de laquelle la correction des sorties de test est garantie par les obligations de preuve dépend de l'approche de modélisation adoptée : dans le pire cas, l'oracle ne peut conclure que lors de l'exécution du code final, car les modèles intermédiaires ne sont pas assez précis vis-à-vis des propriétés attendues. Cependant, ce cas ne correspond pas à la pratique industrielle dans laquelle les propriétés attendues sont exprimées dans une phase de conception préliminaire. Le modèle obtenu à ce niveau est constitué de plusieurs niveaux de raffinement et de décomposition.

Nous avons identifié les conditions à satisfaire par les sous-ensembles d'architecture, afin de pouvoir constituer le modèle correspondant à une étape de développement. Nous avons proposé un algorithme de haut niveau pour le dépliage. Cet algorithme construit le texte formel du modèle déplié, à partir des textes formels des composants B considérés, en mettant à plat les liens existants entre les composants. Cet algorithme n'a pas encore été implémenté. A partir d'un prototype partiel, nous avons cependant précisé quelques détails d'implémentation et identifié des utilitaires de base nécessaires. Notamment, l'existence d'un parseur, permettant de construire l'arbre syntaxique des composants B, est requis. Une étude est en cours à l'INRETS pour développer un module englobant à la fois un parseur (déjà développé) et l'outil de dépliage.

La construction d'un modèle déplié n'est pas seulement intéressante du point de vue du test. Ces modèles permettent d'analyser le comportement induit par une architecture de composants. Ils peuvent donc être utilisés pour la vérification (preuve, model-checking) de propriétés transversales, dont l'expression implique plusieurs composants. La génération de code source peut également se baser sur un modèle déplié, afin de générer du code optimisé.

Dans la suite de nos travaux, nous nous sommes intéressée à la couverture structurelle des modèles B. Toujours dans un cadre théorique unifié, il s'agissait de définir des critères de couverture applicables à la fois aux modèles abstraits et concrets. Or, les critères existants diffèrent selon que l'on cherche à couvrir un programme (analyse du graphe de contrôle) ou une spécification orientée modèle (analyse du prédicat avant-après). Nous avons réalisé l'unification de ces deux types de critères pour les opérations des modèles B, en étendant la notion de graphe de contrôle aux constructions abstraites de la notation (substitutions parallèles, indéterministes, …) et en établissant le lien entre les « chemins » dans ce graphe et la structure propositionnelle du prédicat avant-après correspondant à une substitution B. Nous avons alors défini de nouveaux critères de test, partiellement ordonnés selon leur sévérité, qui viennent compléter la hiérarchie de critères existants. Ces critères peuvent être utilisés à la fois pour la génération de jeux de test structurel, et pour

l'analyse a posteriori de la couverture fournie par des jeux de test fonctionnel. Ils tiennent compte de la notion de bonne définition des modèles B, de sorte que l'analyse structurelle ne génère pas de cas dépourvus de sens.

Pour automatiser la génération des cas de test selon un critère donné, on doit pouvoir distinguer les cas issus de la structure des spécifications de ceux issus de la structure des prédicats de garde. Une solution à ce problème peut être d'extraire les cas de test en parcourant l'arbre syntaxique construit à partir de la spécification d'une opération. Cet arbre syntaxique permet d'identifier les constructions utilisées dans la spécification, et de contrôler ainsi la génération des cas de test définis par chaque critère. Il faudra voir dans quelle mesure ces critères peuvent être implémentés dans des outils existants, par exemple dans l'environnement B-CASTING, éventuellement en proposant des adaptations.

Une extension nécessaire à nos travaux concerne la génération des séquences d'appel aux opérations afin de tester les enchaînements possibles de celles-ci. Chaque opération dans la séquence amène le système dans un état à partir duquel l'opération suivante peut être appelée. Ainsi, l'état encapsulé peut être indirectement commandé et observé. La solution proposée dans la littérature à ce problème est la construction d'un automate à états finis à partir de la spécification, et l'utilisation de critères de couverture de cet automate. Les états de l'automate sont obtenus à partir de l'analyse de partition de l'état du système et les transitions sont des opérations appelées dans un de leurs sous-domaines. La construction d'un tel automate à partir de modèles B dépliés doit être étudiée. Afin de simplifier la construction de l'automate, on peut éventuellement considérer les modèles les plus abstraits, car les obligations de preuve garantissent que les opérations raffinées seront appelées dans la pré-condition la plus externe de leur abstraction.

Mise à part la génération des séquences d'appel aux opérations, les autres axes principaux d'étude pouvant faire suite à nos travaux, sont constitués des points suivants.

La problématique de la génération des entrées de test n'a pas été abordée dans nos travaux. Comme pour le test de programmes, une génération déterministe et une génération statistique peuvent être envisagées. L'imperfection des critères structurels vis-à-vis des fautes que l'on cherche à révéler (par exemple, les fautes dues à une mauvaise compréhension d'un besoin) justifie de considérer la technique de génération statistique, et ce d'autant plus que pour des applications de taille réaliste, seuls des critères faibles sont applicables. Cependant, cette approche nécessite une étude quant à la complexité de l'analyse probabiliste requise pour déterminer une distribution conforme au critère retenu.

La génération de cas de test par nos critères, et la génération d'entrées valuées à partir de ces cas, impliquent la manipulation et la simplification des prédicats de la logique du premier ordre. La logique du premier ordre étant semi-décidable, ces traitements peuvent ne pas être entièrement automatisables. Mais, comme nous l'avons mentionné, ces critères peuvent être utilisés a posteriori pour l'analyse de couverture structurelle des opérations. Pour ce faire, une trace de l'exécution peut être obtenue par instrumentation de l'opération. En ce qui concerne les exécutions

parallèle et indéterministe, un travail théorique doit être cependant effectué pour formaliser l'analyse de couverture de la trace obtenue. Cette analyse suppose que l'animation de l'opération se termine. Comme l'animation procède par évaluation de prédicats valués, elle peut s'avérer plus simple que le calcul de prédicats non-valués.

L'analyse structurelle des spécifications B a les mêmes limites que l'analyse structurelle de programmes. Elle devient vite fastidieuse lorsque la spécification devient complexe. De manière analogue au test de programmes, des stratégies de test d'intégration pour les spécifications B peuvent être envisagées. Par exemple, on peut considérer une approche descendante suivant les étapes de développement : le test du modèle obtenu à une étape donnée peut tenir compte de ce qui a été testé à l'étape précédente. On peut également considérer une approche ascendante en remontant les liens d'importations entre les composants : le test du modèle important plusieurs sous-systèmes peut tenir compte de l'information relative au test de chacun de ces sous-systèmes. La définition de ces politiques d'intégration constitue une suite intéressante à nos travaux, pour laquelle il conviendra de préciser les informations à exploiter dans chacune de ces approches, descendante ou ascendante.

Références bibliographiques

[Abrial 1996] Abrial, J.R., *The B-Book - Assigning Programs to Meanings*, Cambridge University Press, 1996.

[Abrial et Mussat 1998] Abrial, J.R. et L. Mussat, "Introducing Dynamic Constraints in B", *The 2nd International B Conference*, LNCS 1393, pp. 83 - 128, Montpellier (France), Springer Verlag, 1998.

[Alnet 1996] Alnet, S., *Test de programme à partir de spécification B : les problèmes*, Mémoire de DEA, Laboratoire de Recherche en Informatique, 1996.

[Apt 1981] Apt, K.R., "Ten Years of Hoare's Logic: A Survey - Part I", *ACM Transactions on Programming Languages and Systems*, vol. 3, no. 4, pp. 431 - 483, 1981.

[Apt 1984] Apt, K.R., "Ten Years of Hoare's Logic: A Survey - Part II: Non-determinism", *Theoretical Computer Science*, vol. 28, pp. 83 - 109, 1984.

[Back et von Wright] Back, R.J. et Joakim von Wright, *Refinement Calculus - A Systematic Introduction*, Springer-Verlag, 1998.

[Backhouse 1989] Backhouse, R.C., *Construction et vérification de programmes*, Masson, 1989.

[Behm et al. 1999] Behm, P., P. Benoit, A. Faivre, et J.M. Meynadier, "Météor: a Successful Application of B in a Large Project", *World Congress on Formal Methods*, LNCS 1708, pp. 369 - 387, Toulouse (France), Springer Verlag, 1999.

[Behm et al. 1998] Behm, P., L. Burdy, et J.M. Meynadier, "Well Defined B", *The 2nd International B Conference*, LNCS 1393, pp. 29 - 45, Montpellier (France), Springer Verlag, 1998.

[Behm et al. 1997] Behm, P., P. Desforges, et F. Mejia, "Application de la méthode B dans l'industrie ferroviaire", dans *Application des techniques formelles au logiciel,* ARAGO 20, pp. 59 - 87, OFTA – Observatoire française des techniques avancées, 1997.

[Behnia et Waeselynck 1998] Behnia, S. et H. Waeselynck, "External Verification of B Development Process", *The 9th European Workshop on Dependable Computing*, pp. 93 - 96, Gdañsk (Pologne), 1998.

[Behnia et Waeselynck 1999] Behnia, S. et H. Waeselynck, "Test Criteria Definition for B Models", *World Congress on Formal Methods*, LNCS 1708, pp. 509 - 529, Toulouse (France), Springer Verlag, 1999.

[Beizer 1990] Beizer, B., *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, 1990.

[Bernot et al. 1991] Bernot, G., M.C. Gaudel, et B. Marre, "Software Testing Based on Formal Specifications: a Theory and a Tool", *IEE-BCS Software Engineering Journal*, vol. 6, no. 6, pp. 387 - 405, 1991.

[Bert et al. 1996] Bert, D., M.L. Potet, et Y. Rouzaud, "A Study on Components and Assembly Primitives in B", *The 1st Congress on the B Method*, pp. 47 - 62, Nantes (France), 1996.

[Bieman et al. 1998] Bieman, J.M., A.L. Baker, P.N. Clites, D.A. Gustafson, et A.C. Melton, "A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification", *The Journal of Systems and Software*, vol. 8, pp. 13 - 37, 1998.

[Bowen 1996] Bowen, J., *Formal Specification & Documentation Using Z; A Case Study Approach*, International Thomson Computer Press, 1996.

[Bowen et Hinchey 1995] Bowen, J.P. et M.G. Hinchey, "Ten Commandments of Formal Methods: Some Guidelines for Successful Use", *IEEE Computer*, vol. 28, no. 4, pp. 56 - 63, 1995.

[Bowen et Stavridou 1993] Bowen, J.P. et V. Stavridou, "Safety-Critical Systems, Formal Methods and Standards", *IEE-BCS Software Engineering Journal*, vol. 8, no. 4, pp. 189 - 209, 1993.

[Breuer et Bowen 1994] Breuer, P.T. et J.P. Bowen, "Towards Correct Executable Semantics for Z", *The 8th Z Users Workshop*, Cambridge (UK), Springer Verlag, 1994.

[Brinksma 1989] Brinksma, E., "A Theory for the Derivation of Tests", dans *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project,* (P. H. J. van Eijk, C. A. Vissers, M. Diaz, Éds.), pp. 235 - 247, Elsevier Science Publishers North-Holland, 1989.

[Burdy 2000] Burdy, L., *Traitement des expressions dépourvues de sens de la théorie des ensembles : Application à la méthode B*, Thèse de doctorat, Conservatoire National des Arts et Métiers, 2000.

[Carrington et Stocks 1994] Carrington, D. et P. Stocks, "A Tale of Two Paradigms: Formal Methods and Software Testing", *The 8th Z Users Workshop*, pp. 51 - 68, Cambridge (UK), Springer Verlag, 1994.

[Chilenski et Miller 1994] Chilenski, J.J. et S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", *IEE-BCS Software Engineering Journal*, vol. 9, no. 5, pp. 193 - 200, 1994.

[Dehbonei et Mejia 1995] Dehbonei, B. et F. Mejia, "Formal Development of Safety-critical Software Systems in Railway Signalling", dans *Applications of Formal Methods*, (M. G. Hinchey, J. P. Bowen, Éds.), pp. 227 - 252, Prentice Hall Int. (UK) Ltd., 1995.

[Dick et Faivre 1992] Dick, J. et A. Faivre, Automatic Partition Analysis of VDM Specifications, Technical Report, Bull Corporate Research Centre, 1992.

[Dick et Faivre 1993] Dick, J. et A. Faivre, "Automating the Generation and Sequencing of Test Cases from Model-based Specifications", *International Formal Methods Europe Symposium*, LNCS 670, pp. 268 - 284, Springer Verlag, 1993.

[Dick et al. 1989] Dick, J., P.J. Krause, et J. Cozens, "Computer Aided Transformation of Z into Prolog", *The 4th Z Users Workshop*, pp. 71 - 85, Oxford (UK), Springer Verlag, 1989.

[Dijkstra 1976] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, Inc., 1976.

[Duran et Ntafos 1984] Duran, J.W. et S.C. Ntafos, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438 - 444, 1984.

[Floyd 1967] Floyd, R.W., "Assigning Meanings to Programs", *Mathematical Aspects of Computer Science*, vol. 19, pp. 19 - 32, 1967.

[Frankl et Weyuker 1986] Frankl, P.G. et E.J. Weyuker, "Data Flow Testing in the Presence of Unexecutable Paths", *The Workshop on Software Testing*, pp. 4 - 13, Banff (Canada), IEEE Computer Society Press, 1986.

[Frankl et Weyuker 1988] Frankl, P.G. et E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", *IEEE Transaction on Software Engineering*, vol. 14, no. 10, pp. 1483 - 1498, 1988.

[Fuchs 1992] Fuchs, N.E., "Specifications are (Preferably) Executable", *Software Engineering Journal*, vol. 7, no. 5, pp. 323 - 334, 1992.

[Gaudel 1991] Gaudel, M.C., "Advantages and Limits of Formal Approaches for Ultra-High Dependability", *The 6th International Workshop on Software Specification and Design*, pp. 237 - 241, Como (Italie), IEEE Computer Society Press, 1991.

[Gaudel 1995a] Gaudel, M.C., "Advantages and Limits of Formal Approaches for Ultra-High Dependability", dans *Predictably Dependable Computing Systems*, (B. Randell, J. C. Laprie, H. Kopetz, B. Littlewood, Éds.), ESPRIT Basic Research Series, pp. 241 - 251, Springer Verlag, 1995.

[Gaudel 1995b] Gaudel, M.C., "Testing can be Formal Too", *TAPSOFT: Theory and Practice of Software Development*, LNCS 915, pp. 82 - 96, Aarhus (Danemark), Springer Verlag, 1995.

[Gaudel et James 1998] Gaudel, M.C. et P.R. James, "Testing Algebraic Data Types and Process : a Unifying Theory", *Formal Aspects of Computing*, vol. 10, pp. 436 - 451, 1998.

[Gravell et Henderson 1996] Gravell, A. et P. Henderson, "Executing Formal Specifications Need not be Harmful", *IEE-BCS Software Engineering Journal*, vol. 11, no. 2, pp. 104 - 110, 1996.

[Hall 1990] Hall, J.A., "Seven Myths of Formal Methods", *IEEE Software*, vol. 7, no. 5, pp. 11 - 19, 1990.

[Hayes et Jones 1989] Hayes, I.J. et C.B. Jones, Specifications are not (necessarily) Executable, *Software Engineering Journal*, vol. 4, no.6, pp. 330-338, 1989.

[Hazel et al. 1997] Hazel, D., P. Stooper, et O. Traynor, "Possum: An Animator for the SUM Specification Language", *The Joint Asia Pacific Software Engineering Conference and International Computer Science Conference*, pp. 24 - 51, IEEE Computer Society, 1997.

[Hazel et al. 1998] Hazel, D., P. Strooper, et O. Traynor, "Requirements Engineering and Verification using Specification Animation", *The 13th International Conference on Automated Software Engineering*, pp. 302 - 305, IEEE Computer Society, 1998.

[Hierons 1993] Hierons, R.M., *Using Formal Specifications to Enhance the Software Testing Process*, PhD Thesis, Brunel University, 1993.

[Hierons 1997] Hierons, R.M., "Testing from Z Specifications", *Software Testing*, vol. 7, pp. 19 - 33, 1997.

[Hinchey et Bowen 1995] Hinchey, M.G. et J.P. Bowen, *Applications of Formal Methods*, Prentice Hall Int. (UK) Ltd., 1995.

[Hoare 1969] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576 - 583, 1969.

[Hoare 1978] Hoare, C.A.R., "Some Properties of Predicate Transformers", *Journal of Association for Computing Machinery*, vol. 25, no. 3, pp. 461 - 480, 1978.

[Hoare 1995] Hoare, J.P., "Applications of the B-Method to CICS", dans *Applications of Formal Methods,* (M. G. Hinchey, J. P. Bowen, Éds.), pp. 97 - 123, Prentice Hall Int. (UK) Ltd., 1995.

[Hörcher 1995] Hörcher, H.M., "Improving Software Tests Using Z Specifications", *The 9th International Conference of Z Users*, LNCS 967, pp. 152 - 166, Limerick (Irlande), Springer Verlag, 1995.

[Hörcher et Peleska 1994] Hörcher, H.M. et J. Peleska, "The Role of Formal Specifications in Software Testing", *Tutorial Notes for the FME'94 Symposium*, 32 pages, Bruxelles (Belgique), 1994.

[Hörcher et Peleska 1995] Hörcher, H.M. et J. Peleska, "Using formal specifications to support software testing", *Software Quality Journal*, vol. 4, no. 4, pp. 309 - 327, 1995.

[Jones 1993] Jones, C.B., *VDM une méthode rigoureuse pour le développement du logiciel*, Méthodologie du logiciel, Masson, 1993.

[Kans et Hayton 1994] Kans, K. et C. Hayton, "Using ABC to Prototype VDM Specifications", *ACM SIGPLAN Notices*, vol. 29, no. 1, pp. 27 - 36, 1994.

[King et al. 1999] King, S., J. Hammond, R. Chapman, et Pryor A., "The Value of Verification: Positive Experience of Industrial Proof", *World Congress on Formal Methods*, LNCS 1709, pp. 1527 - 1545, Toulouse (France), Springer Verlag, 1999.

[Lano et Haughton 1996] Lano, K. et H. Haughton, *Specification in B: An Introduction Using the B Toolkit*, Imperial College Press, 1996.

[Laprie et al. 1996] Laprie, J.C., J. Arlat, J.P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J.C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, et P. Thévenod-Fosse, *Guide de la sûreté de fonctionnement*, Cépaduès Editions, 1996.

[Loeckx et Sieber 1984] Loeckx, J. et K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons and B.G. Teubner, 1984.

[Mariano 1997] Mariano, G., *Evaluation de logiciels critiques développés pas la méthode B : Une approche quantitative*, Thèse de doctorat, L'université de Valenciennes et du Hainaut-Cambrésis, 1997.

[Marre 1995] Marre, B., "LOFT: A Tool for Assisting Selection of Test Data Sets from Algebraic Specifications", *TAPSOFT: Theory and Practice of Software Development*, LNCS 915, pp. 799 - 800, Springer Verlag, 1995.

[Mikk 1995] Mikk, E., "Compilation of Z Specifications into C for Automatic Test Result Evaluation", *The 9th International Conference of Z Users*, LNCS 967, pp. 167 - 180, Limerick (Irlande), Springer Verlag, 1995.

[Morgan 1994] Morgan, C., *Programming from Specifications*, 2nd Edition, Prentice Hall Int. (UK) Ltd., 1994.

[Myers 1979] Myers, G.J., *The Art of Software Testing*, John Wiley & Sons, 1979.

[Ntafos 1988] Ntafos, S.C., "A Comparison of Some Structural Testing Strategies", *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 868 - 874, 1988.

[OFTA 1997] OFTA, *Applications des techniques formelles au logiciel*, ARAGO 20, 1997.

[Potet et Rouzaud 1998] Potet, M.L. et Y. Rouzaud, "Composition and Refinement in the B Method", *The 2nd International Conference on B Method*, LNCS 1393, pp. 46 - 65, Montpellier (France), Springer Verlag, 1998.

[Py et al. 2000] Py, L., B. Legeard, et B. Tatibouët, "Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes", *AFADL : Approches Formelles dans l'Assistance au Développement de logiciels*, pp. 21 - 35, Grenoble (France), 2000.

[Rapps et Weyuker 1985] Rapps, S. et E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367 - 417, 1985.

[Raynaud et Saint-Gealme 1990] Raynaud, B. et L. Saint-Gealme, "Une approche formelle pour la validation du logiciel", *Troisième journées internationales : le génie logiciel & ses applications*, pp. 699 - 712, Toulouse (France), 1990.

[Roper 1992] Roper, M., "Software Testing: A Selected Annotated Bibliography", *Software Testing, Verification and Reliability*, vol. 2, no. 3, pp. 113 - 132, 1992.

[Rouzaud 1999] Rouzaud, Y., "Interpreting the B Method in the Refinement Calculus", *World Congress on Formal Methods*, LNCS 1708, pp. 411 - 430, Toulouse (France), Springer Verlag, 1999.

[Spivey 1994] Spivey, J.M., *La notation Z*, Méthodologie du logiciel, Masson, 1994.

[Steria 1998] Steria, *Manuel de référence du langage B,* no. Version 1.8.1, 1998.

[Stocks 1993] Stocks, A.P., *Applying Formal Methods for Software Testing*, PhD Thesis, The University of Queensland, 1993.

[Stocks et Carrington 1993] Stocks, P. et D. Carrington, "Test Template Framework: A Specification-based Testing Case Study", *SIGSOFT Software Engineering Notes*, vol. 18, no. 3, pp. 11 - 18, 1993.

[Stocks et Carrington 1996] Stocks, P. et D. Carrington, "A Framework for Specification-Based Testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777 - 793, 1996.

[Taouil-Traverson 1997] Taouil-Traverson, S., *Stratégie d'intégration de la méthode B dans la construction de logiciel critique*, Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, 1997.

[Thévenod-Fosse et Waeselynck 1998] Thévenod-Fosse, P. et H. Waeselynck, "Software Statistical Testing Based on Structural and Functional Criteria", *The 11th International Software Quality Week*, p. 20, San Francisco (USA), 1998.

[Thévenod-Fosse et al. 1995] Thévenod-Fosse, P., H. Waeselynck, et Y. Crouzet, "Software Statistical Testing", dans *Predictably Dependable Computing Systems,* (B. Randell, J. C. Laprie, H. Kopetz, B. Littlewood, Éds.), ESPRIT Basic Research Series, pp. 253 - 272, Springer Verlag, 1995.

[Ural 1992] Ural, H., "Formal Methods for Test Sequence Generation", *Computer Communications*, vol. 15, no. 5, pp. 311 - 325, 1992.

[Van Aertryck 1998] Van Aertryck, L., *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels*, Thèse de doctorat, Université de Rennes 1, 1998.

[Van Aertryck et al. 1997a] Van Aertryck, L., M. Beneviste, et D. Le Métayer, "CASTING : une méthode formelle de génération automatique de cas de tests", *AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, pp. 99 - 112, Toulouse (France), 1997.

[Van Aertryck et al. 1997b] Van Aertryck, L., M. Beneviste, et D. Le Métayer, "CASTING: a Formally Based Software Test Generation Method", *The 1st International Conference on Formal Engineering Methods*, Hiroshima (Japon), IEEE Computer Society Press, 1997.

[Waeselynck 1993] Waeselynck, H., *Vérification des logiciels critiques par le test statistique*, Institut national polytechnique, 1993.

[Waeselynck et Behnia 1997] Waeselynck, H. et S. Behnia, Towards a Framework for Testing B Models, Rapport de recherche, no. LAAS 97225, INRETS-ESTAS/97.29, Laboratoire d'analyse et d'architecture des systèmes, 1997.

[Waeselynck et Behnia 1998] Waeselynck, H. et S. Behnia, "B Model Animation for External Verification", *The 2nd International Conference on Formal Engineering Methods*, pp. 36 - 45, Brisbane (Australie), IEEE Computer Society Press, 1998.

[Waeselynck et Boulanger 1995] Waeselynck, H. et J.L. Boulanger, "The Role of Testing in the B Formal Development Process", *The 6th International Symposium on Software*

Reliability Engineering, pp. 58 - 67, Toulouse (France), IEEE Computer Society Press, 1995.

[West et Eaglestone 1992] West, M.M et B.M Eaglestone, "Software Development: Two Approaches to Animation of Z Specifications", *IEE-BCS Software Engineering Journal*, vol. 7, no. 4, pp. 267 - 276, 1992.

[Weyuker 1982] Weyuker, E., "On Testing Non–Testable Programs", *The Computer Journal*, vol. 25, no. 4, pp. 465 - 470, 1982**.**

Liste des figures

Liste des tableaux

Table des matières